



The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN



Digitized by the Internet Archive
in 2013

<http://archive.org/details/osl2operatingsys460alsb>

OSL/2
AN OPERATING SYSTEM LANGUAGE

by

Peter Allyn Alsberg

CAC Document No. 6



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

The Library of the
JUN 9 1978
University of Illinois
at Urbana-Champaign

Report No. 460

OSL/2
AN OPERATING SYSTEM LANGUAGE

by

PETER ALLYN ALSBERG

June 10, 1971

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

This work was supported in part by the Advanced Research Projects Agency as administered by the Rome Air Development Center under Contract No. USAF 30(602)-4144 and submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, June 1971.

ABSTRACT

OSL/2, an Operating System Language, was designed for the coding of supervisory systems. OSL/2 is an ALGOL-based language that may be readily extended to a general purpose language with the addition of a few data types (e.g., floating point numbers). The language replaces an interrupt concept with the P and V operators on semaphores described by Dijkstra. Concepts of multiple processes and system layers allow OSL/2 to be used as a job control language for an OSL/2 coded supervisory system. The system layering concepts provide for easy system extension and hierarchical structuring. A powerful method of specifying access techniques is used to implement queues, tables, etc. and treat them as primitive data items in the language. The language is presented in narrative and reference formats. Hardware implications are discussed and an OSL/2 compiler which runs on the Burroughs 5500 is briefly described. The implications of the language on system measurement, simulation, and modification are also discussed.

TABLE OF CONTENTS

Chapter	Page
1. OSL/2 Concepts, Philosophies, and Goals	1
2. An Introduction to the OSL/2 Syntax and Semantics	10
2.1. The Primitive Data Types	12
2.1.1. Integer	12
2.1.2. Boolean	13
2.1.3. String	14
2.1.4. Patterns	15
2.1.5. Pointers	16
2.1.6. Conditional and Case Expressions	17
2.2. Composite Data Types	18
2.3. Statements	30
2.3.1. Blocks, Compound Statements, Case Statements and Conditional Statements Are Recursively Defined	30
2.3.2. Iterative Statements	31
2.3.3. Assignment Statement	34
2.3.4. Pattern Matching Statement	36
2.3.5. Procedure Statements	38
2.4. Procedures and Functions	39
2.5. Control Concepts	49
2.5.1. Escape Statements	49
2.5.2. Semaphores	50
2.5.3. Processes	52

Chapter	Page
2.5.4. Traps	54
2.5.5. Primitives	55
2.6. Compile Time Facilities	57
3. System Measurement, Modification, and Debugging in an OSL/2 Environment	60
4. The Hardware Requirements of OSL/2	64
5. Conclusions	78
Appendix	
A. A Reference to OSL/2 Syntax and Semantics	81
B. An OSL/2 Compiler	167
LIST OF REFERENCES	247
VITA	249

LIST OF FIGURES

Figure	Page
1. Variable stack while in block 1	69
2. Variable stack after entering the procedure "addl"	71
3. Two stack layout if " <u>append</u> (addl(r,m))" is written instead of "addl(r,m)".	72
4. Final addressing scheme	75

Chapter 1

OSL/2 Concepts, Philosophies, and Goals

For many years supervisory and monitor systems have been coded in machine level languages by a rather privileged set of system coders. Machine level languages have been used to improve system efficiency. The systems programmer has been allowed the privilege of unprotected access to all portions of a system on the assumption that he is a capable coder not subject to the errors of the normal applications coder.

Unfortunately, assembly language coded systems have shown themselves to be grossly inefficient. For example, consider the differences in system overhead and reliability between two similar systems like ASP [1, 2, 3, 4] and HASP [5] on the IBM system/360. HASP commonly experiences two to four times less overhead than ASP. Yet, in the opinion of this author, a factor of at least 2 if not 4 improvement is still available in HASP if the queuing and event structure were reorganized. Even a reasonably small system like IBSYS [6] for the IBM 7094 has significantly less throughput capability than PORTHOS [7] when a university job mix is being processed. Analyses of the GECOS systems for the GE machines, EXEC systems for Univac machines, etc., would yield similar examples of widely varying efficiencies in different assembly language coded systems for the same machine. By comparison the Burroughs MCP [8] (Master Control Program) for the B5500 experiences relatively low system overhead even though the B5500 MCP is coded in the high level language ESPOL [9]--an ALGOL language derivative. The important point to be made here is that assembly language has not automatically brought efficiencies. Good system design is clearly the most important factor and does lead to efficiencies.

An examination of existing operating systems also shows that, with the notable exception of the THE Multiprogramming System [10], errors are plentiful. The author's personal experience has been that even in very old systems, about five to ten per cent of the errors that were discovered late in the life of a system were undiscovered key punch errors. Of the remaining errors, about one third to one half are addressing errors, most of which could be caught by a simple array bounds or index check in a high level language. After encountering literally hundreds of examples of trivial errors which should not have been made by systems programmers, the author has come to the conclusion that systems programmers are as prone to make mistakes as applications coders. They are in need of tools which help them avoid inadvertant errors by restricting the scope of their actions to the particular task at hand - for example, an automatic index bounds check. Because of the drastic and widely propagated effects of a system error, systems programming requires at least as much protection as that available to applications programming.

The cost of building operating and support systems is now so prohibitive that new machines are required to be downward compatible with old machines on which working assembly language system software already exists. Yet new hardware architectures continue to be developed with a wide variety of addressing features that cannot be used because they conflict with old techniques. This is especially unfortunate when one realizes that the vast majority of system software need not be machine dependent. Most system software is concerned with table and queue manipulation, i.e., with the specification of scheduling algorithms. Machine dependence has been forced as a by-product of the use of machine dependent languages.

The only parts of a system that are truly machine dependent are the interrupt handlers of the supervisor, the code emitter portions of the higher level language compilers, and the assembly language itself.

In 1968 Wells and Alsberg designed OSL - an Operating System Language [11]. OSL executed on one processor and was connected to any number of other processors such as arithmetic units, data channels, and peripheral storage devices. First in first out queues and pointers were provided as primitive data types in the language. Interrupts were handled by automatically invoked procedures. It was possible to dynamically change the procedures to be invoked for each interrupt.

OSL was a block structured ALGOL 60 derivative which interfaced with hardware through eight primitive procedures. For example, the "route" primitive connected an interrupt with the procedure to process the interrupt. Each of these primitives was designed to be implemented on most machines with an extremely small number of instructions (usually, only about a dozen were required). In this manner one could guarantee an easy implementation of the hardware interface on most machines.

A task scheduler for a multiprogramming system was coded in OSL. Since multiprocessing and multiprogramming concepts were embedded in OSL only a few dozen lines of code were required. This code was then examined with the intention of running the same code on an IBM 360/75 and an IBM 7094. Due to the limited time available to the authors, they were unable to build OSL compilers for both machines. However, by the time the project was abandoned, they had verified that the identical OSL code could be made to run on both machines and that the primitives involved could be implemented in just a few instructions on both machines.

OSL was designed with machine independence as a major goal. Thus it was completely divorced from any particular addressing scheme. As a result a multiprogramming system coded in OSL would very likely run poorly on an infinite memory descriptor machine like a B5500 or a virtual memory paged machine like a 360/67 or a Sigma 7. Similarly, a failure to specially treat other machine dependent features such as bulk core, head per track disks, hypertape drives, etc. would lead sometimes to large inefficiencies. However, OSL did assure the transportability of an entire operating system among virtually all medium and large scale computers.

Subsequent work by the author lead to the design of OSL/2. OSL had been designed to provide a common, machine independent systems language for the largest possible number of machines. OSL/2 used another approach. OSL/2 would be machine ignorant in its design. That is, OSL/2 would be designed to easily specify well structured operating system concepts and algorithms independent of existing machines. Once the language was designed the hardware necessary to implement the language concepts would be examined. Finally, the appropriate compromises would be made in the language to make it practical.

Several goals were set at the beginning of the design effort.

- 1) Since the heart of an operating system is its queues and tables, the language must allow the specification and manipulation of arbitrarily structured queues and tables as if they were primitive data items in the language.
- 2) The language should provide facilities for the control and synchronization of multiple processes.
- 3) The specification of hierarchically structured or layered systems must be straightforward.

- 4) It must be easy to insert and delete code to measure the performance of a system coded in OSL/2.
- 5) OSL/2 should be both an operating system language and an adequate job control language. The task of an operating system is to sequence or control one or more processes and to allocate resources to these processes. Similarly a job control language must specify the sequencing or control of one or more processes and specify the resources required by each process. Since these tasks are so similar, one language should handle both tasks. These five goals were met by the process control concepts and the generalized data access techniques.

A process in OSL/2 is a program or procedure that may execute concurrently and asynchronously with any other process. For example, a separately running compiler is a process. It is possible to split the compiler into a scanner and a parser which each run simultaneously. The scanner might process the input stream to find all special words and identifiers and enter these in tables for the parser to use. The parser communicates with the scanner to get the next identifier, special word, or other symbol the scanner has processed. In this case the compiler, a process which includes the parser, would initiate the scanning procedure as a second process. The result is one program with two concurrent processes.

The process management facilities of OSL/2 allow one process to create and, optionally, monitor new processes. If a process initiates a second process which the first process will monitor, then the first process becomes the operating system for the second process. The first process may define new primitives for the second process to use or may even redefine

existing primitives for the second process. For example, a statistical system might create a new primitive that performs correlation and redefine the storage allocation primitive for those processes that will be monitored by it.

Each time a process creates a second process which it monitors, a new level or layer is added to the system. Communication between layers is through the primitive procedures. A primitive action may be frequently redefined in new layers. It is also possible to redefine a primitive within one layer. This will be described in detail later.

A process may spin off one or more processes as asynchronously running procedures. (For example, consider the previous compiler example or the ILLIAC IV operating system [12, 13] which has about a dozen of these processes--one for each operating system task.) When these processes must cooperate with each other, they synchronize their operations through the use of P and V operators on semaphores. These operators were defined and analyzed by Dijkstra [14] and Habermann [15]. P and V operators, which will be discussed later, provide a powerful process synchronization facility. P and V operators are truly primitive in the sense that they are simple to implement in hardware or software and they provide a base from which more complex process communication techniques can be constructed.

The process statements provide system layering and process synchronization facilities. Furthermore, the ability to initiate processes and declare data (e.g. files) at any level, allows OSL/2 to serve as an extremely powerful job control language.

Through the use of the generalized data access facility the programmer may define new data types such as FIFO queues, priority queues,

and special tables. Each time such a data type is accessed, for example in an expression or on the right hand side of an assignment statement, a fetch operation is performed. Each time an attempt is made to store a new value in a data item of this type, for example on the left hand side of an assignment statement, a store operation is performed. The semantics of fetch and store operations and special functions related to the new data type, are defined in the access definition.

The coder can define fetch and store operations for queues or tables that perform very complex manipulations of these data types. Thereafter, a complex process such as inserting a new element into a queue or into a table may be written as a simple assignment statement. If, at a later time, changes are to be made to the queuing algorithms, these changes are made only in the definition and the rest of the code is unaltered. If desired, a coder can add instrumentation in the access definition. Since all manipulations of an access type are defined in the access definition, the definition provides a single spot at which every fetch, store and special function call can be recorded.

Thus the data access facility allows a programmer to define arbitrarily complex data structures that may then be accessed as primitive data items in the language. Furthermore, since this feature is used to implement the queues and tables that define the state of the system, they provide a central place where measurement of the system can be readily performed.

It should be stressed that OSL/2 is not an extensible language in the general sense that GPL [16, 17] and ALGOL 68 [18, 19] are. OSL/2 does allow the coder to specify a class of new data types and the semantic

interpretation of fetches and stores on that data type. Unlike most extensible languages, OSL/2 does not allow binary operators to be created or redefined. This restriction makes the language easier to implement yet still provides those data extension features both necessary and desirable for systems programming.

The author's continual frustration with the lack of good symbol manipulation facilities in existing algorithmic languages led him to adopt the pattern matching concepts of SNOBOL. Many other "nice" features like patterns and case expressions have been added to the language for the convenience of the coder. Most of these facilities are frills that are not essential to the thesis. They are included primarily to document the fact that such formerly diverse constructs as SNOBOL patterns, bit manipulation, and asynchronous processes can all exist harmoniously within a single, block structured, algorithmic language without the specification of scores of new primitives and exception conditions. Most of these facilities were added to basic ALGOL 60 by extending the primitive data base to include structures, strings, patterns and pointers or by making logical extensions to existing statements, such as the iteration statement.

Unfortunately, systems programmers tend to be very poor documenters of their work. Unless they are made to write program logic manuals and flowcharts before they write the code, they will most likely never provide these flowcharts or logic manuals.

Therefore, many of the features in the language are included to aid in the structuring and documentation of a piece of code. For example, the lack of a GO TO statement not only forces clean program structure but also allows a compiler to perform better global optimizations. Since

labels must appear before and after a labelled statement and must be used by the ESCAPE statement, a minimal level of documentation is automatically forced on the coder.

The main features of the language which are essential to the operating system codes are the generalized data access facility which enhances system measurement and queue manipulation and the process facility which allows asynchronous processes to be controlled, synchronized, and layered.

The other features of OSL/2 are not essential but desirable. These extra features make the description of algorithmic processes more pleasant for the coder and frequently make a piece of code more readable. Furthermore, these features not only help the systems programmer structure and compress very complex codes, they also help him clarify his thoughts and thus properly design and structure these arbitrarily complex systems.

Chapter 2

An Introduction to the OSL/2 Syntax and Semantics

The heart of any operating system is its resource scheduling and allocation facilities. Therefore, the highest design priority is placed on the ability of the language to properly specify scheduling algorithms. Secondary to this consideration but still of great importance is the ease with which the programmer can use the language.

Questions concerning paging, file organization, linking, and addressing are not considered in the language.

This chapter is broken into two main sections - a description of the data types and expressions allowed in the language followed by a description of the statements and process control facilities.

The primitive data types allowed in the language are integers, Booleans, strings, patterns, and pointers. Integers are used for arithmetic and indexing. Booleans provide logical manipulation capabilities and are used extensively when controlling program execution. Strings of various character sizes and lengths are used with SNOBOL-like patterns to scan text. Pointers are heavily used to build and process list data or data structured as an arbitrary network.

These five primitive data types are used to form the three composite data types: arrays, structures, and generalized access data. OSL/2 arrays are identical to Algol arrays. Structures allow the specification of tree structured data similar to PL/1 structures.

Generalized access data are built up from the primitive data types and the other composite data types. Special purpose procedures are written to describe how this data will be manipulated. The combination of the data specifications and the manipulation procedures allow one to declare and manipulate complex queues and tables and to perform complex I/O and data retrievals as if they were primitive constructs in the language.

An OSL/2 program is composed of one or more nested or sequential statement blocks. These blocks are identical to ALGOL 60 blocks. Within the blocks, compound, iterative, conditional, and assignment statements are allowed. In addition, procedure calls and case statements are provided. The case statement allows one to index into a list of statements and execute only one of them depending upon which index value was used.

OSL/2 does not have a GO TO statement. The GO TO is replaced by the escape statement. An escape statement allows one to leave the range of any statement he is in. For example, he could leave an iteration loop, or the block that contained the loop, or the block that contained that block, etc. However, one cannot enter another statement via an escape statement. The only way to enter a statement is to arrange for it to be the next statement in the program sequence. In such an environment, a programmer depends heavily upon "infinite" loops, terminated by executing an escape statement, to replace many GO TO statements.

In OSL/2, a process is a separately running program or asynchronously running procedure attached to another process. Control

primitives are available to initiate, stop, and restart processes. Furthermore, it is possible to declare new primitives for processes or to reroute calls on standard primitives to alternative procedures declared by a supervisory process. These asynchronously running processes may synchronize their actions using P and V operators on semaphores. These provide a very convenient lock-out capability.

This brief, global introduction to the language facilities provides the perspective for the detailed description that follows.

2.1. The Primitive Data Types

There are five primitive OSL/2 data types: integers, Booleans, strings, patterns, and pointers.

2.1.1. Integer

The only arithmetic allowed in OSL/2 is performed on the positive and negative integers. In an actual implementation of OSL/2 to solve more than just scheduling problems, it may be desirable to include real numbers or even complex numbers as primitive data. Since these are generally machine dependent and extremely rarely used, if ever, in operating system work, they are not included in the OSL/2 description.

The operations that may be performed on integers include addition, subtraction, multiplication, a variety of divisions, and exponentiation. When performing division on integers, one may produce a rounded or truncated answer. He may further truncate up or down in value. Therefore, division operators are provided to describe each of these operations.

2.1.2. Boolean

Booleans are single valued data that take on the value true or false. They may not be considered to be multivalued or bit strings as they are in most languages. String data and string operators must be used to perform logical operations on bit strings.

The logical functions and, or, not, exclusive or, equivalence, and implication are allowed on Booleans. In the evaluation of Boolean expressions two additional special purpose operations are allowed: andif and orif. The use of these operations is best illustrated with an example.

Assumed that operation X is to be performed if the Boolean variables A, B, and C are all true. An OSL/2 program to do this would be written

if A andif B andif C then X fi

A conditional OSL/2 expression or statement begins with an if and is terminated by a fi. Thus if..fi is actually a bracket pair. In this and all following examples the special OSL/2 words are underlined for illustrative purposes only. In the language itself, words like if, and, etc. are reserved words and may not be used for purposes other than indicating specific OSL/2 constructs. In the above example, one need not test the variables B or C if A is false. The andif caused the evaluation of B and C to be skipped if the expression is already determined to be false. The operator orif causes a similar termination of an expression or subexpression if the value is already determined to be true. These operators are provided for efficiency and to avoid unpleasant side effects. For example, it may happen, particularly in

operating systems codes, that the quantity B may not exist if A is false. Thus the testing of B might cause an error. The way around this situation in conventional languages has been to write nested conditional statements, e.g.,

if A then if B then X fi fi

In OSL this is written as,

if A andif B then X fi

2.1.3. String

String data has a character size and a length. The character size is measured in bits and the length in characters. Strings may be of a fixed length or may be variable within some maximum length.

A string may be bit-wise anded (sand), ored (sor), exclusive-ored (sxor), or complemented (snot). A string may also be tested for a pattern match or for equality or inequality with some other string.

For example, the following Boolean expression is true if the string S contains an "A", "B", or "C".

S { "A" | "B" | "C" }

The pattern is contained within the braces. The following statement tests if the string A is not equal to the string B and, if not, concatenates it to the string X.

if A $\sim$ B then X $\leftarrow$ X & A fi

String inequality is judged by the bit representations of each character. No collating sequences are assumed by OSL/2. However, one may specify standard character sets such as ASCII and EBCDIC when writing string constants. For example, the string constant E"XYZ" is

a three character string. Each character is eight bits long and contains the EBCDIC representation of respectively "X", "Y", and "Z".

Substrings may be specified by following a string expression with a field.

Examples:

$$S.[0:10]$$

$$(S \leftarrow S\&B).[n:k+2]$$

The first example specifies the first ten characters of S. The second example specifies k+2 characters beginning at character n in the new value of S. Characters are numbered left to right beginning at zero.

2.1.4. Patterns

Patterns are used in pattern matching statements. Pattern matching statements are also Boolean expressions. The OSL/2 pattern is based on the SNOBOL 4 pattern.

All patterns are enclosed within patterns to any depth.

Patterns may be assigned to pattern variables and saved for later use or passed to procedures. Patterns may also specify that intermediate results in a pattern match should be assigned to some string variable.

The basic form of a pattern expression is

$$\{A_1 \mid A_2 \mid \dots \mid A_n\}$$

where each A is an alternative. Patterns are matched from left to right. The result of the match is true if one alternative matches and false if no alternative matches. The evaluation of a pattern expression stops at the first match.

Each alternative has the general form

$$S \leftarrow P_1 \&P_2 \&P_3 \&\dots \&P_n$$

where S is an optional string variable and $P_1 \dots P_n$ are pattern primaries. In order to match this alternative, a string must contain the pattern P_1 followed immediately by the pattern P_2 , followed immediately by the pattern P_3 , etc. If a successful match is made, a copy of the part of the string which matches the concatenated patterns is placed in S .

Since a pattern variable may have as its value an embedded assignment to a string variable, one must be sure that the string variable exists so that the embedded string assignment is always legal. Therefore pattern variables or string variables may not appear in a pattern expression on the right hand side of a pattern assignment if the pattern variable on the left is global to them. Furthermore, a formal parameter pattern variable may not be assigned a pattern which contains pattern or string variables. In the example

$$\text{Patt}_1 \leftarrow \{ "A" \mid \text{Str}_1 \leftarrow "B" \ \& \ \text{Patt}_2 \mid \{ \text{Str}_2 \mid \text{Patt}_3 \} \}$$

the following must be true:

1) Patt_2 , Patt_3 , Str_1 , and Str_2 must be declared global to Patt_1 or they must be declared in the same block as Patt_1 .

2) Patt_1 is not a formal parameter.

Pattern primaries are string constants, string variables, pattern variables, pattern functions, and pattern expressions.

Refer to pattern matching statements for more pattern examples.

2.1.5. Pointers

Pointers specify or "point to" any OSL/2 data. If P is a pointer and X some data, then $P.X$ is the X pointed to by P . This is of special use when list processing. The OSL/2 coder will normally

have only one identifier used to denote a list element. The pointer then denotes which list item is being specified. Pointers may point to node, own, or normal data although they are primarily intended for use with node data (node data may be explicitly allocated and deallocated by the programmer).

Pointers may point to other pointers. For example, "P1.P2.X" is the X pointed to by the P2 which is pointed to by P1.

Pointers may have a null value or may be assigned values via allocate statements and pointer functions. $P \leftarrow \text{ptr}(N)$ creates a pointer to the variable N and assigns it to P.

2.1.6. Conditional and Case Expressions

In the previous five data types, the following primaries are allowed in all expressions.

if Boolean expression then expression₁ else expression₂ fi

and

case integer expression of

expression₀,

expression₁,

⋮

expression_n esac

In the above constructs, expression₁ is an integer, Boolean, string, pattern, or pointer expression.

In the first construct, the if primary, the Boolean expression is evaluated, if it is true, then the value of the primary is expression₁. If it is false, the value of the primary is expression₂.

Examples:

if x=0 then a else b fi

n ← if n=0 then 4 else n+1 fi

The case primary has as its value expression_i where i is the value of the integer expression. If the integer expression is less than zero or greater than n then the default expression_n is the value of the primary.

2.2. Composite Data Types

There are three composite data types, arrays, structures, and generalized access data.

OSL/2 allows any of the primitive or composite data types to be subscripted. The subscript list follows the data identifier and is enclosed in brackets.

Examples:

A[0]

B[10,23*2+7]

C23[if X_is_even then X else 2*X fi]

Data structures, similar to the structures allowed in PL/1, are used in OSL/2. An example of a list element declaration is given below.

```
node structure list_element
(structure data (integer age, weight, height;
                Boolean sex_is_male);
 pointer forward, backward)
```

Note that structures may be nested within structures. Also note the use of an underscore to make identifiers more readable. The underscore is a null character. Thus the identifiers "a_b_c" and "abc" are the same identifier.

The data within a structure must always be completely addressed by all identifiers in the hierarchy above the desired data.

The list of identifiers is in hierarchial order and is separated by

".". For example, the pointer forward is addressed as

```
list_element.forward
```

and the integer weight is addressed as

```
list_element.data.weight
```

"list_element.weight", "weight", "data.weight", and "forward" would not be legal syntax in any construct.

If desired, for example when passing data to a procedure, one may specify a whole structure or substructure by addressing down to the level desired. For example, "list_element" or "list_element.data".

The generalized data access facilities allow the programmer to specify the action that is to be taken when data is fetched from (variable appears on the right hand side of an assignment statement) or stored into (variable appears on the left hand side of an assignment statement) the variable. In addition, numerous side actions may be specified. This facility is of greatest use when manipulating tables, queues, and stacks which have complex data structures and access algorithms. In access declarations and definitions, the key words access, stack, table, and queue may all be used interchangeably.

For a simple example consider a queue consisting of integers which are accessed in a first in first out manner such that storing into the queue causes a new entry to be made and fetching causes the oldest entry to be deleted.

Two constructs are required: the definition of the data type and a declaration of an identifier which is this type. First the definition:

```

1. queue fifo;
2. begin
3.     integer (number_of_entries, oldest_entry, newest_entry)  $\leftarrow$  0,
4.         entry_list[0:99];
5.     integer fifo procedure fetch;
6.     begin
7.         fetch  $\leftarrow$  entry_list[oldest_entry  $\leftarrow$  (*+1)mod 100];
8.         number_of_entries  $\leftarrow$  *-1
9.     end fetch;
10.    fifo procedure store (X); integer X;
11.    begin
12.        entry_list[newest_entry  $\leftarrow$  (*+1)mod 100]  $\leftarrow$  X;
13.        number_of_entries  $\leftarrow$  *+1
14.    end store;
15. end

```

A line by line description of this definition will provide insight into many OSL/2 constructs.

1) On this line the new access type "fifo" is declared. Within the scope of this definition, the identifier "fifo" will be used to declare "fifo" queues.

2) All the information concerning an access type is local to the access variable. This prevents tampering with access variables through any means other than the access procedures declared in the access definition. Thus the definition block is a true block enclosed in a begin and end (lines 2 and 15) that restricts the use of internal access variables.

The access procedures declared within the definition are pseudo local to the type fifo. That is, it is possible to write the OSI/2 code "length (X)" in one place and have it represent a call to one of many procedures called length depending on the declaration of "X". "length(X)" might refer to a standard procedure call with one actual parameter.

If "X" is an access variable and an access procedure, "length", is declared for that access type, then the appropriate access procedure will be invoked. If several access types are available each with its own procedure named "length", the particular procedure "length" declared to be part of the access type of the access variable will be invoked.

3) Three integers are declared and are initialized to zero.

4) An integer array indexed from zero to 99 is declared.

5) This line declares the fetch procedure. The fetch procedure will automatically be invoked whenever a fetch is made on a variable declared to be fifo. The result will be an integer.

6) The fifo procedure fetch has a compound statement as a body. Thus, it is enclosed in a begin end pair.

7) The return value of the function "fetch" is set here. The value returned is in the array "entry_list". The index of "entry_list" is an imbedded integer assignment statement. The index will be the value of "oldest_entry" after the assignment is performed. Notice the use of the asterisk in the right hand side of the assignment statement. This is a shorthand for the variable used on the left hand side of the assignment statement. Only one such asterisk is allowed and it must be the first integer quantity (primary) encountered on the right hand

side. Thus

$$A \leftarrow * + 1$$

is the same as

$$A \leftarrow A + 1$$

but

$A \leftarrow 1 + *$ is not allowed. Note that this expression increments "oldest_entry" and uses the mod operator to map its values cyclically over the range zero to 99.

8) This line is unnecessary now but will be used later. It maintains a count of the number of entries currently in the fifo queue. The use of the asterisk convention is illustrated here to decrement the number of entries in the queue.

9) This end closes the fetch procedure compound statement. Note that comments consisting of letters and digits are allowed following an end.

10) The store procedure is declared on this line. A store procedure is automatically invoked when an assignment is made to an access variable. Store procedures have one parameter which is the type of the data which may be stored into the access variable. The type of data fetched from and the type of data stored into an access variable need not be the same type. The identifiers "fetch" and "store" indicate that the fifo procedures with those names have special restrictions as to their automatic invocation and number of parameters. In this case the data stored into the fifo access variable is integer data.

11) This begin starts the compound statement which is the body of the procedure "store".

- 12) In this case, an entry will be made in "entry_list". Otherwise, this is similar to line 7. Note that, as in line 7, an embedded assignment is used to compute the subscript of "entry_list". This position in the array receives the value to be stored.
- 13) In a manner similar to line 8, the "number_of_entries" variable is incremented.
- 14) As in line 9, this line terminates the procedure store and also contains a comment.
- 15) This line terminates the fifo definition.

In order to use fifo variables, one must declare them. For example:

```
fifo queue X, Y, Z
```

declares three fifo variables X, Y, and Z.

Some examples of use follow:

$X \leftarrow 0$	puts a zero on the tail of X.
$Y \leftarrow Y$	removes the head of Y and puts it at the tail of Y.
$Y \leftarrow Y \leftarrow Y \leftarrow 1$	puts 3 ones into Y.
$Y \leftarrow Z \leftarrow X$	removes the head of X and puts it on the tail of Z and Y.
$Y \leftarrow Y \leftarrow Y \leftarrow Y$	is equivalent to $Y \leftarrow (Y \leftarrow (Y \leftarrow Y))$.

Note that subscripts and all other explicitly stated operations in an assignment statement are evaluated from left to right. However, assignments are made right to left. Thus the invocations of access store procedures, which occur at assignment time, are right to left.

Now that the basic access data type has been demonstrated, assume that the coder wishes to be able to add a facility to his fifo queues to enable him to find the length of the queue. The following access procedure declaration may then be added to the definition between lines 4 and 5, 9 and 10, or 14 and 15.

```
integer fifo procedure QSIZE;
```

```
QSIZE ← number_of_entries;
```

With this addition, the coder can write QSIZE(X) and know the number of entries in X. Note that the parameter "X" ("X" is a fifo queue variable) does not appear in the declaration of "QSIZE". All access procedures are called with one more parameter than is declared. By convention, this additional parameter is the first parameter and indicates which access variable is to be operated upon.

Some additional features of the access facility are demonstrated by this example:

```
1. stack simple(S); integer S ← 0;
2. begin
3.     integer number ← 0, default_value ← S;
4.     pointer head ← null;
5.     node structure entry(
6.         integer item;
7.         pointer next ← head);
8.     integer simple procedure fetch;
9.     begin
10.        pointer temp;
11.        if head = null
12.        then leave fetch (default_value)
13.        else
```

```

14.          fetch ← head.entry.item;
15.          temp ← head;
16.          head ← head.entry.next;
17.          release (temp.entry)
18.      fi
19.  end fetch;
20.  simple procedure store(X); integer X;
21.  begin
22.      head ← allocate(entry);
23.      head.entry.item ← X
24.  end store
25. end;
26. simple stack X, Y(A*B);
27. X ← 1;
28. Y ← X ← A+B;
29. X ← X;
30. X ← X ← X;
   :
   :

```

Note that in line 1 the access type "simple" has a formal parameter which is used on lines 3 and 12 to return a default value if the simple stack is empty.

Note also that on line 5 a node structure is declared to hold the entries in the stack. Node structures, like PL/1 controlled storage, may be allocated and deallocated by explicit programmer command (cf. the allocate and release statements on lines 17 and 22).

On line 7 of this definition, the substructure pointer "next" will be initialized to the value of "head" each time the structure entry is allocated.

Lines 12 and 14 illustrate the two ways to return values for a function. On line 12, the escape statement "leave fetch (default_value)" sets the function value and exits. On line 14, the function value is set but computation continues.

When specifying a node variable, if a pointer to the variable is not given, the last allocated instance of that variable is assumed. Thus on lines 14, 16 and 23, the phrase "head" need not appear before the phrase "entry".

On line 26, the simple stacks "X" and "Y" with respective defaults of zero and A*B are declared. In the definition of simple stacks on line 1, the parameter "S" was given a value of zero which was to be used if the parameter was not supplied at declaration time. Since the parameter was omitted in the declaration of "X", zero was assumed.

One might add the following access procedure and internal procedure between lines 19 and 20.

```

19.1      integer procedure glorp(n,p); value n,p; integer n, pointer p;
19.2      begin
19.3          if p = null then leave glorp (default_value)
19.4          else if n=0 then leave glorp (p.entry.item)
19.5          else leave glorp (glorp (n-1, p.entry.next)) fi
19.6          fi
19.7      end glorp;
19.8      integer simple procedure scan(n); value n;
19.9          integer n ← 0;
19.10     scan ← glorp(n,head)

```

This declares the procedure "scan" which may be used to non destructively access any item in the stack. Note that procedure "scan"

has an optional parameter "n" which is assumed to be zero, the top of the stack, if it is not supplied by the coder (lines 19.8 and 19.9).

The recursive procedure "glorp" is local to the "simple" stack definition block and may only be used by the procedures within the definition block. The procedure glorp does the actual search through the stack for procedure "scan".

"scan(X)" and "scan(X,0)" will both return a value equal to the top element on the stack X. "scan(X,n)" will return element "n" or the default value (zero in this case) of the stack if the stack does not contain an nth entry.

Access definitions and procedures may redefine the interpretation of subscripts. One may follow the formal parameters of an access definition with array parameters. The parameters that appear in the array part need not be arithmetic - they are handled the same as any formal parameter.

Example

```
access formatted_file (A,B,C)[X1:X2,X3,X4:X5:X6];
integer A,B,C,(X1,X2,X3,X4,X5,X6) ← 0;
      :
```

The array parameters in the access definition define the interpretation and format of the array part of an access declaration. The array parameters may be declared and defaulted in the same manner that other formal parameters are. The array part parameters are then input in the same fashion as in any other declaration.

Example

```
formatted_file access(F1,F2)(30,30,150),
      F3(30,150,150)[0:10],
      (F4,F5,F6)(30,30,150)[0:2,13,10:2:20],
      FA(30,150,150)
```

Subscripted access variables may appear in OSL/2 code. In each case the appropriate fetch, store, or special access function must have a subscript specification.

Example

definition:

```

integer A[0:N,0:N];
access transpose
begin
    integer transpose procedure fetch [B,C]; integer B,C;
    fetch  $\leftarrow$  A[C,B];
    transpose procedure store (D)[B,C]; integer B,C,D;
    A[C,B]  $\leftarrow$  D;
end

```

declaration:

```
transpose X
```

use:

```
X[N,M]  $\leftarrow$  A[N,M]
```

The above access example produces a variable X which, when used as an array, produces the transpose of the matrix A.

The array facilities in access declarations and definitions allow one to interpret an array subscript as a particular queue or table entry or to imply virtually any other interpretation. Since the number of subscripts in the declaration and the format of the definition need not agree and can be defaulted, a maximum amount of freedom is allowed.

For example, one could declare a special array that took advantage of a subscript declaration with this form: "A:B:C" where an array would be created with indices from "A" to "B" in steps of "C".

A fifo fetch procedure with the following heading could be written.

```
integer fifo procedure fetch [N]; integer N ← -1;
      :
```

One could use the array subscript to specify a destructive or non-destructive fetch from the queue if N were -1 (no subscript) then a destructive fetch of the oldest element would proceed. Otherwise, the N^{th} element would be non-destructively fetched.

In general the access procedure heading

$$\text{Pr}(A_1, A_2, A_3 \dots A_n)(N_1, N_2, \dots, N_m);$$

would be called as

$$\text{Pr}(\text{Vr}[N_1, N_2, \dots, N_m], A_1, A_2, \dots, A_n)$$

where defaults may arbitrarily trim the actual number of N's or A's that appear and Vr is a particular access variable name. The actual parameters, both N's and A's, may be integers, booleans, strings, etc. In particular, the subscripts (N's) need not be arithmetic and need not be interpreted as normal subscripts.

Access variables can be declared to make complicated table entries, perform formatted input/output, and monitor variables.

Perhaps one of the greatest advantages of the access mechanism is the ease with which the coder can use it to measure a system. If one uses the access mechanism to perform all stack, queue, and input/output operations, a centralized place is provided to insert instrumentation. One merely adds to the access definition that code which is desired. It is then easy to keep statistics on virtually every access variable.

Furthermore, it is easy to delete any measurement code once it has served its purpose.

2.3. Statements

2.3.1. Blocks, Compound Statements and Conditional Statements Are Recursively Defined.

That is, a single statement may contain one or more statements embedded within it. Each of these embedded statements may in turn contain further embedded statements, etc.

This happens in one of two ways. First, assignment statement may be embedded in expressions which are themselves parts of statement. Second, many statements may be grouped together within a block, compound statement, case statement, or conditional statement.

If one places several statements end to end, separates them with semicolons and then puts a begin in front and an end at the rear he has a single compound statement. This compound statement is then treated as one statement, not the several of which it is composed. This is especially important when parsing case statements and procedure declarations. The compound statement has the following basic form:

begin S; S; S; S; S; S;...S; S; S end

If one takes the statement list previously described, precedes it with a list of declarations separated by semicolons, and then wraps it in a begin end pair, he has a single block--which is also a single statement. The basic form of a block is

begin D; D; D; S; S; S;...S; S end

A case statement and a conditional statement take the same forms as case and conditional expressions (cf. 1.1.6.) except that they select from a choice of statements rather than expressions. Conditional and case statements have the following forms:

if Boolean expression then S; S;...; S else S;...; S fi
case integer expression of S₀;...S_{n-1}; S_n esac

2.3.2. Iterative Statements

Iterative statements take many forms. They are used to control loops. One may step through a loop until some condition is satisfied, incrementing some index, or choosing items from a list. The forms of the iterative statement are generally self-explanatory. In the following forms several abbreviations are used. V is a variable, S a statement, I an integer, B a Boolean, Str a string, Ptr a pointer, Pat a pattern, and E an expression. For example, B.E. is a Boolean expression and I.V. an integer variable.

Iterative Statement Forms:

- 1) while B.E. do S
- 2) do S until B.E.
- 3) for I.E. times do S
- 4) for V $\leftarrow$ E₀, E₁, ..., E₂, do S

In form 1, the Boolean Expression is evaluated and tested.

If it is true, the statement following the do is executed and the whole process begins again with a reevaluation of the Boolean expression.

If it is false, the statement is skipped and control continues after the iterative statement.

Form 2 is similar to form 1 except that the statement is executed first. Then the Boolean expression is evaluated. In this case, a false result continues the loop and a true result terminates the loop.

Form 3 simply specifies that a statement is to be unconditionally executed a fixed number of times.

Examples:

while i < j do

```

begin
    i ← i+1;
    some_function (i,j)
end
do A[i ← *+1] ← 0 until i = n
for n times do factorial ← *(i ← *+1)

```

Form 4 actually has many variations. The variable V may be integer, Boolean, string, pointer, or pattern. The types of the expressions $E_0, \dots, E_n$ are the same type as the variable V . In the basic form 4, the variable V is first assigned the value E_0 . Then statement S is executed. Then V is assigned the value E_1 . Then statement S is executed again. This continues through all of the E 's.

If the variable in form 4 is an integer, some interesting variations are allowed in the E 's. In this case the form of the E 's may be

- 4.1). I.E.
- 4.2). I.E.₁ to I.E.₂
- 4.3). I.E.₁ to I.E.₂ by I.E.₃
- 4.4). I.E.₁ step I.E.₂ until B.E.
- 4.5). I.E. while B.E.

Form 4.1 is identical with a standard form 4 iteration. Form 4.2 specifies that the variable will be stepped from I.E.₁ to I.E.₂ by the increment +1. If I.E.₁ > I.E.₂ no statement will be executed and the next E in the form 4 iteration expression list will be executed. Form 4.3 is the same as 4.2 except that an increment is specified. In this case I.E.₃ may be positive or negative and the test for the end of this iteration expression is then made respectively on $V > I.E._3$ or $V < I.E._3$.

Form 4.4 specifies that V is to be set to $I.E._1$. The Boolean expression is then tested. If it is true this iteration expression is finished. If it is false then the statement S is executed; $V \leftarrow V + I.E._2$; and the Boolean expression tested again. Form 4.5 specifies that $V \leftarrow I.E.$. If the Boolean expression is false, no statement is executed--the iteration expression is finished. If the Boolean expression is true, the statement S is executed; $V \leftarrow I.E.$; and a new test is made.

In the evaluation of the integer expressions above, bad side effects can present themselves. For example, the loop end, $I.E._2$, in form 4.2 may be a complex expression that takes significant time to compute each time through the loop. Often times, systems coders desire these effects. However, it is necessary to carefully evaluate each expression and assign it to a separate variable outside the loop if extra computations are to be avoided. Therefore, in each of the forms 4.1 to 4.5 one may write value ($I.E.$) instead of $I.E.$. If this is written, the $I.E.$ is evaluated only once, at the beginning of each iterative expression, rather than many times (i.e. once for each iteration) for each iteration expression.

Examples:

1. for $S [I] \leftarrow "IF", "NOT", "MAYBE"$ do $S2 \leftarrow *S[I]$
2. for $x \leftarrow 1, 2, 3, n$ step 1 until $\text{sqrt}(x) < 100*n, 15, 16,$
 i to value ($j*f(n)$) by 11, $x+1$ while $x < n$ do $\text{print}(x, y, z)$
3. for $x \leftarrow x+1$ while $x < 10$ do
begin
 $\text{if even}(x)$ then $x \leftarrow x+1$ fi;
 $\text{test}(x)$
end

2.3.3. Assignment Statement

The assignment statement assigns a value to a variable. These variables may be integers, Booleans, strings, pointers and patterns. Furthermore whole structures and substructures can be assigned new values from another structure or substructure.

The basic assignment statement has the form

variable $\leftarrow$ expression

or

structure $\leftarrow$ structure expression.

The assignment arrow may be either the symbol " $\leftarrow$ " or ":=".

Since expressions can also be assignment statements, the general form of the assignment statement is

$V \leftarrow V \leftarrow V \dots V \leftarrow V \leftarrow E$

The evaluation of primaries and subscripts is made from left to right subject to the conventional precedence of operators. The order of assignment is right to left. The order of assignment should normally only be important if access variables appear on the left hand side of an assignment arrow.

In the case of integer, Boolean, and string assignments, an asterisk convention is allowed. That is, the presence of an asterisk as the very first primary to appear on the right of an assignment arrow denotes the same variable as the variable to the immediate left of the assignment arrow. For example, let x and n be integers and A an array.

$x \leftarrow * + 1$ is equivalent to $x \leftarrow x + 1$

$x \leftarrow (**n) \dots$ is equivalent to $x \leftarrow ((x*n) \dots$

$A[x] \leftarrow *+n$ is equivalent to $A[x] \leftarrow A[x] + n$

$n \leftarrow x \leftarrow * + n$ is equivalent to $n \leftarrow x \leftarrow x + n$

$A[x \leftarrow * + 1] \leftarrow * + n$ is NOT equivalent to $A[x \leftarrow x+1] \leftarrow A[x \leftarrow x+1] + n$

it IS equivalent to $A[x \leftarrow x + 1] \leftarrow A[x] + n$

Note in this last example that the side effect of incrementing x when evaluating the subscript of A is not duplicated on the right. The asterisk notation refers to the identical variable as appears on the left of the arrow not to the same variable expression as appears on the left.

When assigning values to pointers, the values so assigned must be at the same or higher level as the pointer receiving the value. Thus, if P_1 is a pointer declared in a block that is global to the block in which P_2 is declared, $P_1 \leftarrow P_2$ is not allowed but $P_2 \leftarrow P_1$ is. All formal procedure parameters are considered global to any parameters declared in the outermost block.

Patterns must also obey the same rules of assignment as pointers.

Structures or substructures may receive the values of other structures or substructures provided that there is a complete match of all data types in the respective structures or substructures.

Example:

begin

structure A (integer B,C; structure D (integer A));

structure B (integer X,Y; structure Z (integer L));

structure C (integer A);

A $\leftarrow$ B;

A.D $\leftarrow$ B.Z;

A.D $\leftarrow$ C;

C $\leftarrow$ B.Z

end

This is a correct program. Note that $A \leftarrow C$ would not be allowed since A and C have different type structures.

2.3.4. Pattern Matching Statement

The pattern matching statement is the basic SNOBOL string replacement statement.

Form: $S[\underline{\text{fixed}}] P [\leftarrow \text{string expression}]$

S is a string variable, P a pattern expression and optional items are enclosed in the metalinguistic square brackets. A pattern matching statement is also a Boolean expression which is true if a match is made.

Several examples would give a quick feel for the power and use of pattern matching statements.

1. Does the string S contain the letter "A", "B", or "C"?

$S \{ "A" \mid "B" \mid "C" \}$

2. Does S begin with "A", "B", or "C"?

$S \underline{\text{fixed}} \{ "A" \mid "B" \mid "C" \}$

Note that the reserved word fixed specifies that the patterns must match beginning at the very first character.

3. Does S contain two "A"'s?

$S \{ "A" \ \underline{\&\text{any}\&} \ "A" \}$

This pattern match is true if the string S contains the letter "A" followed by any pattern followed by another "A".

4. Replace all occurrences of the letter "A" in string S with the digit "1"

$\underline{\text{while}} \ S \{ "A" \} \leftarrow "1" \ \underline{\text{do}}$

This uses an iteration statement with a dummy statement. The intent is to repetitively execute the pattern matching statement until a match is no longer made. When a match is made, that part of the

string matched is replaced by the string expression to the right of the arrow.

5. Some strings begin with the character "(" . If the character ")" occurs later in the string, replace the "(" and ")" by "[" and "]" .

```
s fixed {"("&{Stemp ← any&")"} ← "["& Stemp & "]"
```

In this case a string variable, "Stemp", has been used to store the string matched by the pattern any. "Stemp" thus saves the characters between the parentheses for later use in the string assignment to the matched pattern.

6. Replace the first occurrence of "A", "B", or "C" in "S" followed by any single character with the same "A", "B", or "C" followed by a period.

```
S {S1 ← {"A" | "B" | "C"} & any(1)} ← S1 & "."
```

This is another instance of saving a partial string for use later in the assignment. Note the optional parameter on the any pattern. This is a pattern that matches any string of characters of length "1".

7. The following pattern assignment creates a pattern that will recognize an identifier.

```
ID ← {span ("ABCDEFGHJKLMNOPQRSTUVWXYZ")  
&{span ("ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789") | MT}}
```

The pattern variable "ID" may now be used to scan the string "CARD" to find the first identifier and store the identifier into string "S".

```
CARD {S ← ID}
```

Assume that the string "CARD" contains a list of identifiers that should be placed in the array "x". The following codes would all assign the successive identifiers beginning at "x[1]".

```
integer I ← 1;
```

```
while CARD {any & {x[I] ← ID}} ← MT do I ← I + 1
```

or

```
integer I ← 0;
```

```
for I ← I+1 while CARD {any & {x[I] ← ID}} ← MT do
```

or

```
integer I;
```

```
for I ← 1 step 1 while CARD {any & {x[I] ← ID}} ← MT do
```

In the above examples, note that a pattern match is attempted on the first part of the string--anything followed by an identifier. If a match is made, the statement is true and the matched part of the string "CARD" is replaced by the empty string. Thus the first identifier is removed from the string so that further tests are not made on previously recognized identifiers. The assignment to the array "x" is performed as a side effect of the pattern match. It is common when using pattern matching statements as the termination condition of a loop, that the statement following the do is empty.

2.3.5. Procedure Statements

Procedure statements specify that certain, previously declared, portions of code are to be executed. The OSL/2 procedure statement is similar to the ALGOL 60 procedure statement and similar to the FORTRAN subroutine call.

The procedure statement consists of the procedure identifier followed by an optional parameter list enclosed in parenthesis.

Examples:

```
X
```

```
Resequenece
```

```
Call_Subr (A,B,C)
```

```
Rewind (tape_9)
```

2.4. Procedures and Functions

OSL/2 procedures are declared and used in a manner very similar to ALGOL 60 procedures. The primary difference is that default parameters are allowed in procedure and function calls.

A procedure declaration consists of a procedure heading followed by a statement. For example, a simple procedure called "add_one_to_A" might have the following declaration.

```
procedure add_one_to_A;
```

```
A ← * + 1
```

The later occurrence of the statement "add_one_to_A" in the OSL/2 program will cause the statement "A ← * + 1" to be invoked.

The procedure heading may specify some formal parameters to the procedure body (the statement following the procedure heading).

Example:

```
procedure add_one (x);
```

```
integer x;
```

```
x ← * + 1
```

In this example the formal parameter "x" is provided for use in the procedure body "x ← *+1". "x" must be declared in the procedure heading. The procedure heading lists all the formal parameter identifiers. This list is enclosed in parentheses and follows the procedure name. Each formal parameter must be declared in the formal parameter specification. The formal parameter specification is part of the procedure heading and precedes the procedure body.

The occurrence of the statement "add_one(A)" later in the OSL/2 program will now invoke this procedure with "A" substituted for

"x" in the procedure body. Thus "add_one(A)" will cause the statement "A ← *+1" to be invoked.

Formal parameters may be called by name, by value, or by reference. If they are called by value, their identifiers must appear in the value part of the procedure heading. The value part precedes the formal parameter specification. Formal parameters which are called by name must have their identifiers appear in a name part in the procedure heading. The name part may take the place of, immediately precede, or immediately follow the value part. Parameters which are neither called by name nor called by value are called by reference. The general form of a procedure heading is

```
procedure <procedure id>[(<id> ,... , <id>)];
[value <id> ,... , <id> ,;]
[name <id> ,... , <id> ;]
[<declaration> ; ... <declaration> ;]
```

Where the square brackets are metalinguistic and indicate optional syntax. The positions of the name and value parts may be interchanged.

Example:

```
procedure add_3(A,B,C,N);
value N; integer A,B,C,N;
begin
    A ← *+N;
    B ← *+N;
    C ← *+N
end
```

In this example, the formal parameters "A", "B", and "C" are called by

reference and "N" is called by value. The statement "Add_3(X,Y,Z,L+M+Q)" is equivalent to

```
begin
    integer Ninternal;
    Ninternal ← L+M+Q
    X ← *+Ninternal;
    Y ← *+Ninternal;
    Z ← *+Ninternal
end
```

If the formal parameter "N" was called by name the statement "add_3(X,Y,X,L+M+Q)" would be equivalent to

```
begin
    X ← *+L+M+Q;
    Y ← *+L+M+Q;
    Z ← *+L+M+Q
end
```

When one calls an actual parameter by name inside the procedure body, the entire actual parameter, with parenthesis around it if necessary, replaces the formal parameter. When one calls an actual parameter by value, the actual parameter expression is calculated once upon entry to the procedure. This calculated value is stored in a variable local to the procedure and all references to the formal parameter are made from and to this local variable.

Call by value allows one to increase efficiency by eliminating many evaluations of the identical expression and allows one to avoid undesirable side effects. For example, if the actual parameter "N" was

" $L \leftarrow L+M+Q$ " in the above example and if "N" was called by name, the equivalent statement to " $\text{add\_3}(X,Y,Z,L \leftarrow L+M+Q)$ " would be

```
begin
     $X \leftarrow *+(L \leftarrow L+M+Q);$ 
     $Y \leftarrow *+(L \leftarrow L+M+Q);$ 
     $Z \leftarrow *+(L \leftarrow L+M+Q)$ 
end
```

Note that the value of "L" is continually changed. This may be a bad side effect if unexpected.

Call by value may be used to keep an external variable from being changed by a procedure call.

Example:

```
procedure Add_N(A,N);
value N; integer N,A;
while  $N \leftarrow N-1 \geq 0$  do  $A \leftarrow A+1$ 
```

While this may be the hard way to add "N" to "A", it does illustrate how one would protect the value of "X" in the statement " $\text{Add\_N}(X,X)$ ". If "N" is called by value, this will double "N" (assuming "N" is positive). If "N" is called by name, this statement would be equivalent to "while $X \leftarrow X-1 = 0$ do $X \leftarrow X+1$ " which is an infinite loop.

The procedure "Add_N" also demonstrates the illegal statement side effect. " $\text{Add\_N}(X,23+16)$ " is legal if "N" is called by value. If "N" is called by name, this is equivalent to "while $(23+16) \leftarrow (23+16)-1 \geq 0$ do $X \leftarrow X+1$ ". This clearly has an illegal assignment to " $(23+16)$ " which is not a variable.

If a parameter is not specified to be called by name or called by value, then it is called by reference. Parameters called by reference are similar to FORTRAN subroutine parameters. If possible a "reference" is a pointer to or an address of a single variable, a structure, an array, a substructure, or a single element of a structure or an array. This "reference" (pointer or address) is calculated once, just before procedure entry. If it is not possible to generate such a "reference" then the actual parameter will be called by value. This is generally the case when a complex expression or procedure call is used as a parameter.

Example:

Assume the following global declarations

```

structure S (integer X,Y);
integer I,L,M,N;
integer A[0:10];
structure Big (integer X,Y [0:10]);
procedure P (I,A,S);
    integer I,A[0:10];
    structure S (integer A,B);
    begin
    :
    end P

```

Let us call the procedure "P" with the following actual parameters for "I".

S.X	}	These parameters will have references calculated. Note that each subscript will be computed only once.
A[I]		
A[I ← I+1]		
Big.S.Y		
L		
N		

L+N+M	}	These parameters will be reduced to call by value parameters.
I ← I+1		
A[I]+A[I+1]		
S.X*N		

Let us call "P" with the following actual parameters for the formal parameter "A"

A	}	References will be calculated for these arrays.
Big.Y		

Let us call "P" with the following actual parameters for the formal parameter "S"

S	}	References will be calculated for these structures.
Big.S		

<u>if</u> I < L <u>then</u> S <u>else</u> Big.S <u>fi</u>	}	These structure expressions will be called by value.
<u>case</u> I <u>of</u> S, Big.S, Big.S <u>esac</u>		

In systems programming, one frequently calls a procedure with many parameters such that one or two of these parameters are varied and the others are usually the same. OSL/2 allows one to specify a default

parameter for a formal parameter by initializing the formal parameter in the formal parameter specification.

Example:

```

integer standard_priority ← 25,
        standard_core ← 15000,
        standard_time ← 3600,
        standard_lines ← 10000;
    :
procedure insert_in_job_Q (Job,priority,core,time,lines);
    value Job,priority,core,time,lines,
    structure Job (integer type, steps_completed;
                    Boolean TTY_Job);
    integer priority ← standard_priority,
        core ← standard_core,
        time ← standard_time,
        lines ← standard_lines;;
    begin
    :
    end

```

When one calls a procedure, he may specify default values by leaving the actual parameter position empty, inserting an asterisk, "*", or by eliminating the parameter position if it is the last part of the parameter list. The following calls are then all equivalent:

```

insert_in_job_Q (J[23])
insert_in_job_Q (J[23],*,,)
insert_in_job_Q (J[23],,standard_core,*,*)
insert_in_job_Qn(J[23],standard_priority, standard_core,
                  standard_time, standard_lines)

```

When an initialization appears in a formal parameter specification, an actual parameter equal to the initialization is constructed. This is saved and inserted at any later procedure statement when a default value for this parameter is desired. The specification of a default parameter is a flag to the compiler. It does not generate any additional code in the procedure body.

To further aid the specification of parameters, a key word facility is provided. A particular actual parameter may be input to a procedure by default, by placement in the appropriate position of the ordered actual parameter list or by specifying the formal parameter identifier followed by a colon and the actual parameter. Keyword actual parameters need not appear in any particular order. They do however, occupy one position each in the ordered actual parameter list. Therefore, if a keyword actual parameter is used, the parameter position it occupies must be specified elsewhere as a keyword actual parameter or it must be defaulted.

Examples based on the previous procedure "insert_in_job_Q":

```
insert_in_job_Q (J[23], Time: standard_time, standard_core)
insert_in_job_Q (core: standard_core, standard_priority,
                  Job:J[23])
```

```
insert_in_job_Q (J[23], priority: standard_priority)
```

All OSL/2 procedures are reentrant. In order to allow one pass compilation, all procedures and data must be declared before they are used. This would cause a significant problem if one had a procedure A which invoked procedure B and B could also invoke A recursively. The "forward" procedure declaration is used to provide the information the compiler

needs to compile calls on a procedure without compiling the procedure body of A which contains the call on the as yet undeclared procedure B. In the forward procedure declaration, the procedure heading is complete, but the procedure body is replaced by the reserved word "forward". The complete declaration of the forward procedure must follow later with a statement for the procedure body.

Example:

```

procedure A(N); value N; integer N; forward;
      :
procedure B(M,N); integer N,M;
begin
      :
      A(M);
      :
end;
      :
procedure A(n); value N; integer N;
begin
      :
      B(Q,R)
      :
end;
      :

```

Occasionally, a procedure is a formal parameter to a procedure.

In this case, the complete procedure heading must be provided in the formal parameter specification. However, the reserved word "formal" will be used as the procedure body.

Example:

```

procedure add (A,B,C); integer A,B,C; A  $\leftarrow$  B+C;
procedure sub (A,B,C); integer A,B,C; A  $\leftarrow$  B-C;
procedure either (operation, X,Y,Z);
      integer X,Y,Z;
      procedure operation (X,M,A); integer X,M,A; formal;

```

```

        operation (X,Y,Z);
integer (L,M,N) ← 0;
        either (add, L,L,1);
        either (sub, L,M,N);

```

A function is a procedure that can return an integer, Boolean, string, pointer, pattern, or structure value. Functions may thus be used as primaries in expressions.

A function declaration has a type designation preceding a procedure declaration. Any substructure information or string parameter information must be part of this type designation.

Example:

```

integer procedure a(X,Y,Z);...
Boolean procedure b(X,Y);...
string procedure c(x);...
string (N,M) procedure d(X,y);
string (length1) procedure e(x);...
pointer procedure f(x,y);...
pattern procedure g;...
structure (integer A,B;
        structure X(pointer left, right; string name);
        structure Y(integer A,B,C);
        Boolean b1,b2) procedure Svalue (x,y);...

```

To leave a procedure body, one either executes the entire body and exits at the last statement, or he executes an escape statement. In a procedure the escape statement is simply the reserved word leave

followed by the procedure name. When leaving a function, the escape statement may have an expression enclosed in parentheses following it.

Examples:

```
leave sqrt(23)
```

```
leave sfl("ABC")
```

```
leave Calc (A+B/16)
```

These escape statements return a value for the function as they exit the procedure body.

It is possible to have the procedure identifier appear on the left hand side of an assignment statement within the procedure body. This assignment statement assigns an interim value to the function. If, later, an exit is made via the end of the procedure body or with a simple escape statement (not followed by an expression) then the last such interim value assigned is returned as the function value.

Example:

```
integer procedure add (A,B); integer A,B;
```

```
add ← A+B;
```

2.5. Control Concepts

2.5.1. Escape Statements

The go to statement has been excluded from OSL/2. In its place is the escape statement "leave", which exits one or more statements.

The escape statement used to exit a procedure body has already been described.

The labelled leave statement is used to exit a statement. In order to leave a statement, a label must be provided in front of and after the statement. A colon separates the label from the statement.

```
form: label: statement :label
```

The leave statement is the reserved word leave followed by a label.

Example:

```

outter:  while a<b do
inner:  begin
        .
        .
        leave outter;
        .
        .
        leave inner;
        .
        .
end :inner :outter

```

The occurrence of a label after a statement is required for documentation purposes only. The first label determines the parse.

The label in a leave statement may be global to the block or procedure body in which the escape statement appears.

2.5.2. Semaphores

Semaphores were introduced by Dijkstra [14] for the synchronization of parallel processes. Habermann [15] has proven that P and V operators on semaphores are sufficient to safeguard any critical sections of system code.

A semaphore is a special purpose integer variable. Semaphores are declared analogously to integers. The reserved word semaphore replaces the reserved word integer.

Example:

```
semaphore a,b,c,(d,e,f) ← 1,x,y,z ← 0,q,r,s
```

Semaphores are normally initialized to zero unless otherwise specified in their declaration.

The only operations that may be performed on semaphores are P and V (written "P(s)" and "V(s)" where s is a semaphore).

When a process performs the operation $P(s)$, the semaphore s is decremented by one ($s \leftarrow s-1$). If the decremented value of s is negative ($s < 0$) then the process is blocked and is booked on a waiting list.

When a process performs the operation $V(s)$, the semaphore s is incremented by one ($s \leftarrow s+1$). If the value of s is positive ($s > 0$) then no further action is taken. If the value of s is non-positive ($s \leq 0$) then one or more processes were blocked and are booked on a waiting list. In this case one process is removed from the waiting list and may now proceed.

P and V operators can be used to implement complex waiting queues and priorities. Therefore, the primitive queuing algorithm is not specified. Regardless of the original queuing algorithm on a single semaphore, a new one can be built through the use of one or more semaphores.

P and V provide access to critical code or some system resource. To implement any scheduling scheme on x , one could introduce a process X which schedules x . A process which desires x would then use one semaphore to pass this request to X and another to block its continuation until x was available.

Example:

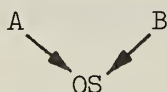
$P(msg);$	seize message path to X
$\vdots$	send message to X requesting x
$V(msg);$	release message path to X
$P(x_i)$	seize x
$\vdots$	use x
$P(msg);$	seize message path to X
$\vdots$	send message to X releasing x
$V(msg);$	release message path to X

Note in the above example that each process would use the "msg" semaphore to communicate with X but each process would use a different semaphore (indicated by x_i) to block its access to x. The process X would perform any necessary $V(x_i)$ to permit a particular process (the i^{th} process to which semaphore x_i refers) to have access to x. Clearly, X can implement an arbitrary booking and dispatching algorithm for x.

2.5.3. Processes

OSL/2 processes are separately running, asynchronous procedures or programs. Six primitives control these processes: spawn, append, initiate, suspend, restart, and terminate.

The spawn primitive allows one to spawn, from inside a process, another process which will run at the same level as the spawning program. The spawning program and spawned program have no special relationship. If the already executing process A of operating system OS spawned process B the relationships would be diagrammed as follows.



Any parameters passed to the spawned process are called by value if they are defined within the spawning process.

Examples:

spawn (program)

spawn (routine (x,y,z))

A process may append a procedure. An appended procedure has access to all normally addressable constructs. A process may not leave a block until all procedures appended in that block have returned. Appended procedures run asynchronously with the appending process.

Example:

```

begin
    integer (A,B,C)[0:9], X,Y,Z,N ← 9;
procedure inner_product (A,S); integer A[0:N],S;
begin
    integer sum ← 0,i;
    for i    0 to N do sum ← sum + A[i]*A[i];
    S ← sum
end  ;
:
:
    append (inner_product(A,X));
    append (inner_product(B,Y));
    inner-product(C,Z);
end

```

In this example three processes would be running simultaneously.

Two new processes would be created to perform the procedure "inner_product" on arrays "A" and "B", and the original process would calculate "inner_product" of "C". All three processes have access to the same global variable "N" but each inner product procedure has its own local variables "i" and "sum" which are not addressable by the other processes.

Initiate creates a new process that is to be monitored by the initiating process. The new process may not directly address constructs within the initiating process. All communication to the initiating process must take place through the primitives defined by the initiating process and any parameters passed to the process. A process variable is attached to each such process. The process variable is used to "suspend", "restart", and "terminate" the initiated process.

Example :

```
process p1,p2,p3;
initiate (program, p1);
initiate (program, p2);
initiate (user job (10,file_23),p3);
suspend (p1);
suspend (p2);
restart (p2);
terminate (p3) ;
```

If a terminate statement appears in a process without a process variable for a parameter, then the process in which the statement is executed is terminated.

Example:

```
terminate
```

2.5.4. Traps

Traps are special conditions that arise in the course of executing an operator. For example, divide by zero or index out of range. If desired a procedure may be executed to handle the special condition.

Examples:

```
on index terminate
on overflow leave overflow (99999)
on divide_by_zero (Numerator) leave divide_by_zero (Numerator)
```

Traps vary from machine to machine. Therefore, each OSL/2 implementation must define reserved or key words for each such trap. The trap declaration

consists of the reserved word on followed by the trap condition followed by an optional set of parameters followed by a statement. The parameters contain pertinent information about the environment. For example, an implementation might return the value of an index causing an index fault and also the array bounds involved. While index out of range, integer overflow, and divide by zero are clearly required in OSL/2, the definition of the trap environment (i.e. trap parameters) is left to each implementation.

The trap facility allows a way of specifying special operators for a microcoded machine or perhaps a way of changing the semantics of existing operators. For example, one implementation might allow a trap on all integer divides so that the coder could define the semantics of divide.

2.5.5. Primitives

The reinterpretation of the primitives between system layers is frequently implementation dependent. While OSL/2 specifies the manner in which reinterpretation will be performed the actual specification of a primitive may vary from installation to installation.

In general, the scope of primitive declarations is the same as for normal procedure declarations with the following extensions. Within the declarations at the head of a block, all access to a primitive made lexicographically prior to the new declaration of a primitive are valid and will access the old primitive. Furthermore, the types of primitive functions must be maintained. Therefore, it is not permitted to change ptr (x) from a pointer primitive to a string primitive.

Example:

```

begin
    integer X[0:10000];
    pointer primitive allocate (.....);
    :
    integer A[0:100]
end

```

In this example, an implicit call on the old primitive "allocate" is performed to allocate the array "X". Note that the reserved word primitive is used in place of the reserved word procedure when redefining a primitive. Once "allocate" is redefined, all allocation will be performed by the new primitive. Thus the array "A" will be allocated by the new primitive. Since the only space addressable by the primitive "allocate" is the array "X", "A" must be allocated inside of "X". Furthermore, any processes which are appended or initiated (but not spawned) within the scope of the new allocate must use the new allocate.

The number and type of the parameters into which a compiler will map an allocate statement is not defined. OSL/2 originally had a data type "core" into which allocations could be made. This facility was deleted because it restricted the class of addressing schemes allowed in the hardware. In order to permit the maximum flexibility for future technological developments, such as cheap content addressed memories, the manner of allocation and the format of the parameters are left undefined.

Most of the arithmetic function primitives will not be changed. The allocate, release, and terminate primitives might be in a normal system. The primary use of the primitive construct will be to define

additional primitives for use at higher levels in a multi-layer operating system.

2.6. Compile Time Facilities

Operating system codes have a strong need for a parameterized macro facility and some conditional compilation. These facilities allow a significant isolation of changeable parameters and they also allow a programmer to write and document a single system that has several, possibly conflicting, portions that are conditionally inserted on an installation by installation basis.

The define facility used in Burroughs B5500 [23] and B6500 [24] ALGOL is used in a slightly modified form in OSL/2. One may define a string value for an identifier.

Examples:

```
define ABC = begin
```

```
    A ← 0;
```

```
    B ← N;
```

```
    C ← M
```

```
    end #;
```

```
define Xsize = 100#,Ysize = 37#;
```

The "=" and "#" bracket the string. The appearance of the defined identifier later in the block in which it is defined will cause the identifier to be replaced by the characters enclosed in the "=" "#" brackets.

In the above example, $A \leftarrow Xsize$ is equivalent to $A \leftarrow 100$.

Defined identifiers may appear in defines nested to any level.

Furthermore, defines may appear within defines. "define identifier=...#" is considered a bracket construct that may appear in a define.

Defines may have parameters.

Examples:

```
define add (A,B) = ($A)+($B)#;
```

```
define concat (A,B) = "$A$B" #;
```

In the above cases, add (10*L,13) is equivalent to (10*L)+(13) and concat (ABC,DEF) is equivalent to "ABCDEF".

The dollar sign (\$) denotes compile time facilities. One may declare compile time Booleans, integers, and strings.

Example:

```
$integer A,B,C,D ← 10,E;
```

```
$string (S1,S2) ← MT;
```

These variables must be preceded by the dollar sign when they are used. An assignment to a compile time variable is made at the time it is scanned. The expression assigned must therefore be calculable at compile time. A compile time variable which appears in an expression is replaced by a constant whose value is the same as the current value of the variable.

Examples:

```
$A ← $D+1;
```

```
X ← 10 + $A*$D; is equivalent to X ← 10 + 11*10;
```

Most OSL/2 statements may be compile time statements.

Examples:

```
$if $A<10
```

```
then ...
```

```
else ...
```

```
fi
```

```
$for $A ← 1,13,2 do X[P(QRZ),$A] ← L*13
```

In both of the above examples, the \$if and \$for did not require the "\$". In the if case, the expression being tested was a constant, therefore, a good compiler would only have compiled one branch of the conditional anyway. In the case of the for statement, the use of a compile time variable as the iteration control variable indicated that the statement was a compile time statement. The for statement is equivalent to:

$X[P(QRZ),1] \leftarrow L*13;$

$X[P(QRZ),13] \leftarrow L*13;$

$X[P(QRZ),2] \leftarrow L*13$

while...do..., do...until..., and for...times do... always must be preceded by a dollar sign if they are compile time statements.

Chapter 3

System Measurement, Modification, and Debugging in an OSL/2 Environment

Operating systems are not static. They are constantly being changed. These changes either improve the efficiency of the system, add features to it, or correct errors.

When new systems are coded, the most significant question facing the implementor is: "How well does it run?" This question is answered predictively by a closed form analysis of the system or by simulation. Once implemented, the question is answered by measuring the system.

Due to the complex interaction of system modules, and the stochastic nature of the inputs a closed form prediction of the performance of an entire operating system is usually not possible. However, many individual modules in a system may be amenable to some greater or lesser degree of analysis.

Simulation is more commonly used when predictive estimates of an entire system's performance are desired. Unfortunately, it is usually very difficult to accurately estimate the loads on a system. If one assumes, however, that an accurate model of the system is available, then it is certainly reasonable to run this model in a variety of modes and under a variety of loads. In this manner, the areas of competence and incompetence of the system can be discovered a priori. This information can then be used by the managers of an installation to make the loads placed on a system fall in the area of the operating system's greatest competence.

The strict isolation of processes and the nature of the synchronizing primitives P and V in OSL/2 make OSL/2 a reasonable simulation

language. Therefore, it is possible to avoid a separate simulation step by using OSL/2 for both the simulation and the final system code.

OSL/2 has no concept of an interrupt. Systems coded in OSL/2 are command driven. That is, they consist of separate processes for each task to be performed. Each task uses the P and V operators to get and dispatch work. Therefore, each process is either waiting for a command to begin work (P operation), issuing a command to another task to begin work (V operation), or performing its function.

Since OSL/2 systems have well defined interfaces to both the outside world and internal processes, it is a relatively straightforward (although frequently tedious) procedure to simulate an OSL/2 coded system in total or in part by appropriately setting global variables and issuing P and V operations to the various semaphores of the system.

The generalized data access facilities of OSL/2 allow access to every transaction performed on any data declared with this facility. Therefore, one may specify virtually any information he wants recorded on access data, the manner of recording, and the manner of read out. Since this is done only once, in the access definition, a single place is provided where all measurements of queues, stacks, tables, etc. may be inserted and deleted. This facility allows the system designer, subject only to the constraint of the Heisenberg Uncertainty Principle, to readily measure an OSL/2 coded system a priori during simulation or a posteriori after implementation.

Changes to an OSL/2 coded operating system may take place in several ways. The generalized data access facilities provide an easy way to change queuing algorithms and add special features to complex data

structures. One may add new facilities by adding a new primitive to an existing system or by introducing a new layer to the system and adding the additional feature in the new layer. This latter technique is especially desirable if one has a well functioning piece of code which he does not want disturbed. Adding the new layer to the system, leaves the original system undisturbed. Furthermore, the new layer is transparent to layers above which are using the primitives in the older, lower system.

Since OSL/2 code can express only nested loops and procedures and cannot express interlocking loops with deliberate transfers to various portions of code and returns via indexed and non indexed go to's, the code tends to be more clearly structured and therefore easier to debug. However, it is still possible to write bad code and design bad logic. OSL/2 does not attempt to solve the coding or logic error problem. The generalized data access facilities provide a sophisticated data monitoring tool of considerable power. However, no special debugging facilities have been added to the language.

Balzar has done some excellent work with EXDAMS [20] on a 360 PL/1 system. EXDAMS provides a method of keeping a history of a program's execution. After a run is completed, this history is coupled to a graphic display which is used to selectively examine variables and run the program backwards as well as forwards to discover where errors lie. This history has links between the execution variable and time space and the source program space. These links are used to immediately see which program statements caused the actions indicated on the graphic display. OSL/2 is amenable to this treatment as are most higher level languages.

In general OSL/2 appears to be a powerful language for coding, expanding, and measuring an operating system. While OSL/2 tends to be cleaner and thus easier to debug, special debugging features have not been provided.

Chapter 4

The Hardware Requirements of OSL/2

It was the original intention of the author to design and simulate an OSL/2 machine. That is, a machine especially suited to executing OSL/2 code. This project was not completed for three reasons. First, the amount of machine time required to build and run a simulator was not available. In fact, the author was requested to sharply reduce the machine time spent building an OSL/2 compiler. Second, the human time required to design the OSL/2 machine in detail and to code a simulator was significantly underestimated by the author. Third, since the author had gone through the exercise of designing the language with an actual implementation in mind, it is doubtful that the completion of an implementation would significantly alter the concepts in OSL/2.

The author actually wrote OSL/2 code, wrote an OSL/2 compiler, and designed the language with a particular hardware implementation in mind. The coding of a compiler and the resulting object machine considerations had their primary impact on the syntactic description of the language. Similarly, the production of OSL/2 code changed the syntax again. The only concept significantly altered was the ability to pass whole declarations into access definitions as parameters to be supplied at access variable declaration time. The intent was to allow the programmer to describe a data independent access technique and introduce the actual types of data after the definition of the access type. For example, this would enable a programmer to define a first in first out queue and at declaration time describe the data being entered and removed from the queue.

This proved to be quite difficult to implement. Later codes showed that this feature was still desirable but the language was not badly hurt by its absence.

The author had had experience with a large number of machines of first through fourth generation architecture. The ease with which one could compile code for Burroughs machines and the similarity between higher level language constructs and the machine operations in the B5500 and B6500 originally prejudiced the author toward those designs.

The OSL/2 compiler assumed that the object machine, like a B6500, was a zero address stack machine with each word in memory tagged to indicate the type of its contents. Furthermore, a block level relative addressing scheme with arrays indexed via B6500-like descriptors was assumed [21]. This made the object code relatively easy to produce. However, the author soon discovered that the isolation of types (i.e. they could not be mixed in expressions) and the explicit coding of type transfer functions made the use of tagged words to distinguish types unnecessary. Furthermore, since OSL/2 procedure declarations require the complete specification of all parameters, the compiler can check at compile time for the erroneous passing of parameters and procedures to procedures. Thus the compiler knows at all times exactly which types it is working with. In this case, most of the type tags on words were necessary only to protect the system from compiler errors and not to aid the compiler.

Virtually all of the OSL/2 constructs can be implemented on any machine upon which FORTRAN may be implemented. The exceptions are call by name procedure parameters and the process control primitives.

With the exception of the Burroughs compilers, call by name has been a problem since it was first introduced in ALGOL 60. The normal solution has been the "thunk". Since call by name parameters are evaluated each time they are referenced in a procedure, a procedure call, or "thunk", is substituted for the reference. Then each time a store or fetch is to be made, this procedure is executed to produce an address or value. On the B5500 and B6500 a hardware operand call is made. The tag bits of the operand tell the computer whether a simple fetch or procedure entry and complex expression analysis must be performed to produce a value. The operand call instruction is much more complex than the most intricate instructions on conventional machines. It does, however, replace a "thunk" with a single hardware operator of much greater efficiency.

Call by name is a concept absent from most languages. It is included in OSL/2 for two strong reasons. It allows the use of side effect processes in a simple way. The concept of a continuously evaluating expression for use in calculating process priorities is an example of how this feature could be well used in operating system codes. Also, it is the nature of access data that they produce complex side effects when they appear in expressions. A call by name concept is required if expressions containing access variables are to be passed to procedures.

In order to produce an efficiently running OSL/2 machine it is considered essential that the hardware provide a call by name facility. Such a facility seems to be easier to implement on a stack machine than a normal multiregister machine. This is because of the ease with which evaluation procedures may be automatically entered and the process environment saved in the stack. To enter a procedure and save the process

environment is a more difficult matter in a register machine where the process environment may be determined by an arbitrary number of the registers.

The problem of inter layer communication is severe. No hardware known to the author is capable of both allowing access to variables in the system at all levels, yet rigorously assuring that variables are not inadvertently altered by an erroneous compiler or operating system. Ordinary relocation and bounds protect or virtual memory addressing will not work.

If some layer of the operating system makes a mistake while setting the bounds register or relocation register--all is lost. In a similar manner, a virtual memory system might have an error made while its segment table was set up. However, a few extraordinary techniques help solve these problems. For example, the register loading or segment address loading instruction could be made a special operator that worked only with an already set relocation and bounds register or an existing segment table. The operator could be designed to not create any segment address or register setting that would violate the existing settings. Thus one could guarantee that higher layers of the system would not inadvertently bomb lower layers if the lower layers properly initiated the upper layers. Techniques like these can be used to remove the concept of privileged instructions from a machine. Each operator is constructed so it has a built in check that prevents it from violating its own environment. Thus any operator (instruction) in the machine can be allowed at any level.

Where this scheme falls on hard times is in the passing of parameters between system layers. How does system layer three pass a file

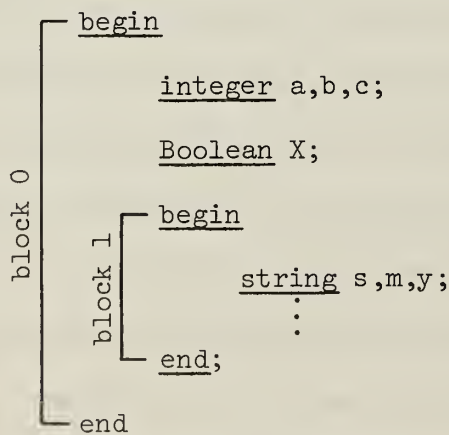
and an integer to system layer ten for its use without allowing access to all of layer three's variables? The setting of protection bits clearly gets immediately out of hand if layer 3 frequently gets control of layer 10. Then the setting and resetting of protection bits becomes a very significant overhead.

The author feels that the solution to these problems lies in a suitably defined addressing scheme. An addressing scheme where it is simply not possible to generate an address outside of one's allowed space.

The addressing scheme will be described in several steps. First in terms of a single program, then in terms of a program with several asynchronous processes, and finally in terms of a multilayered operating system.

A stack machine is assumed with several level registers. Each level register points to the base address of a block level in an OSL/2 program.

Example:



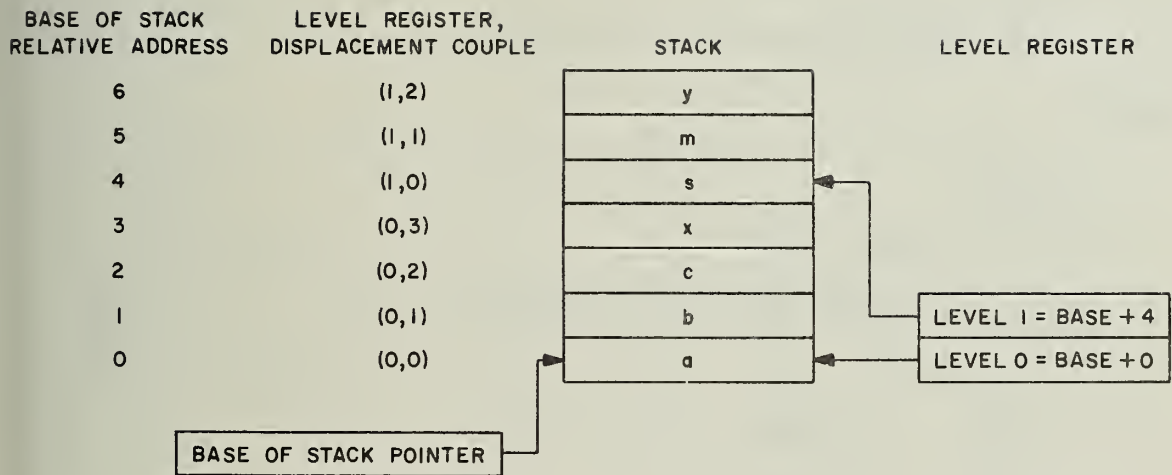
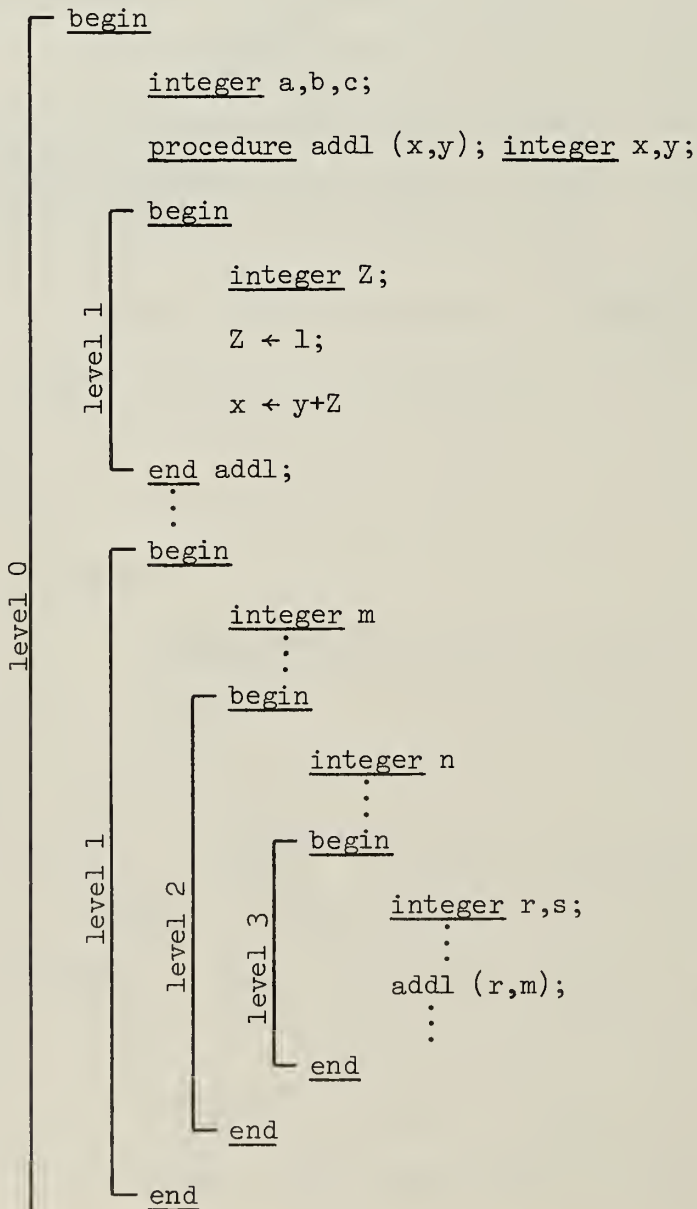


Figure 1. Variable stack while in block 1.

In order to address the variable "a" in block 0 the address couple (0,0) would be used - level 0, displacement 0. To address the variable "c", the address couple (0,2) is used - level 0, displacement 2. Similarly, the variable "m" in block 1 is (1,1) - level 1, displacement 1. One could even put bounds on each level register to prevent inadvertent addressing into the next higher level. Then one would have

a set of miniature relocate and bounds registers. Actually, this is not necessary but desirable. In fact, the entire program stack is an array with bounds protect and a base of stack pointer. Furthermore, the distinction between levels is made more clear by inserting special data into the stack at each break in a level to indicate the old value of a level register. This is necessary when doing recursion. This additional data is used to pass parameters to procedures.

Example:



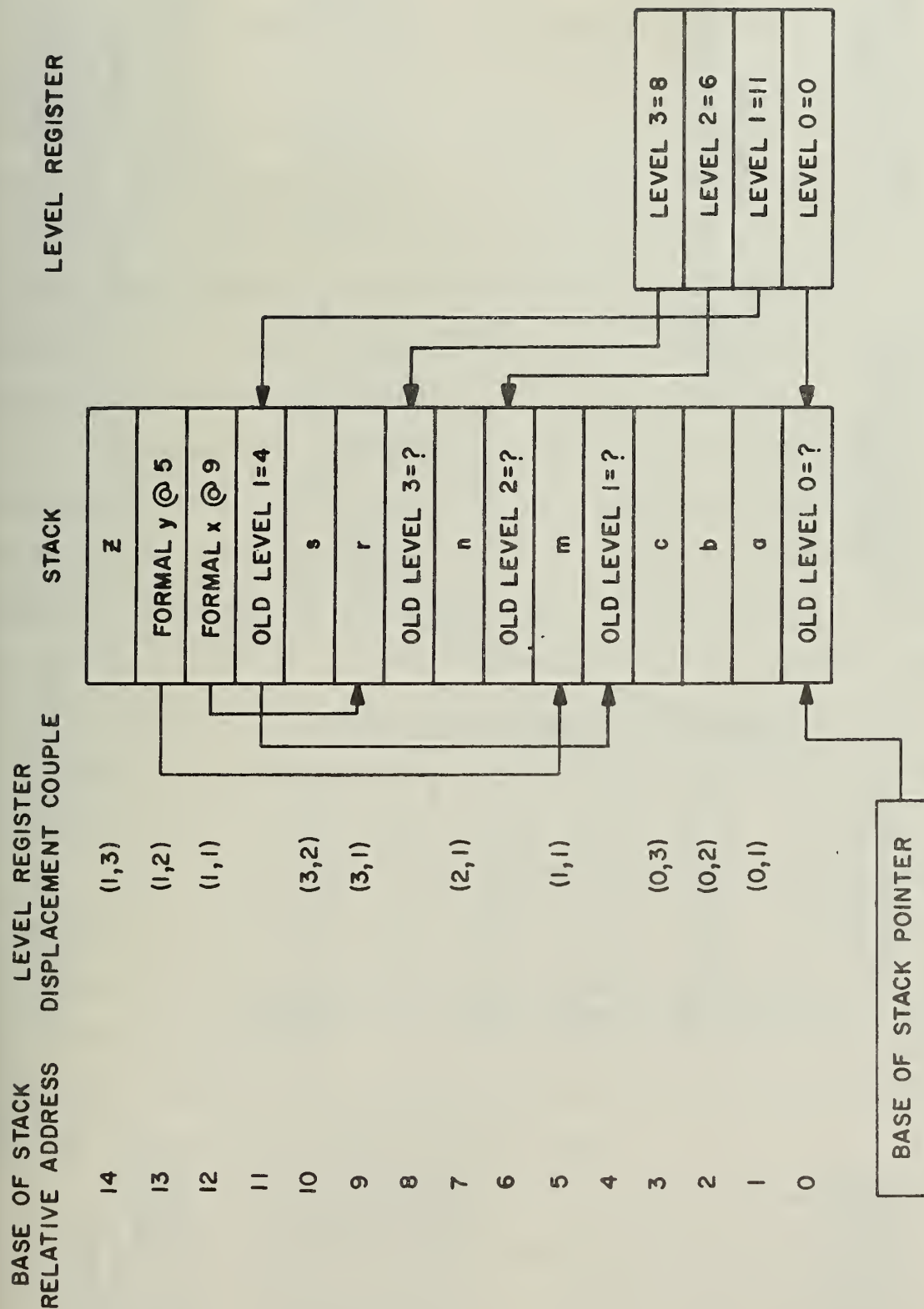


Figure 2. Variable stack after entering the procedure "add1".

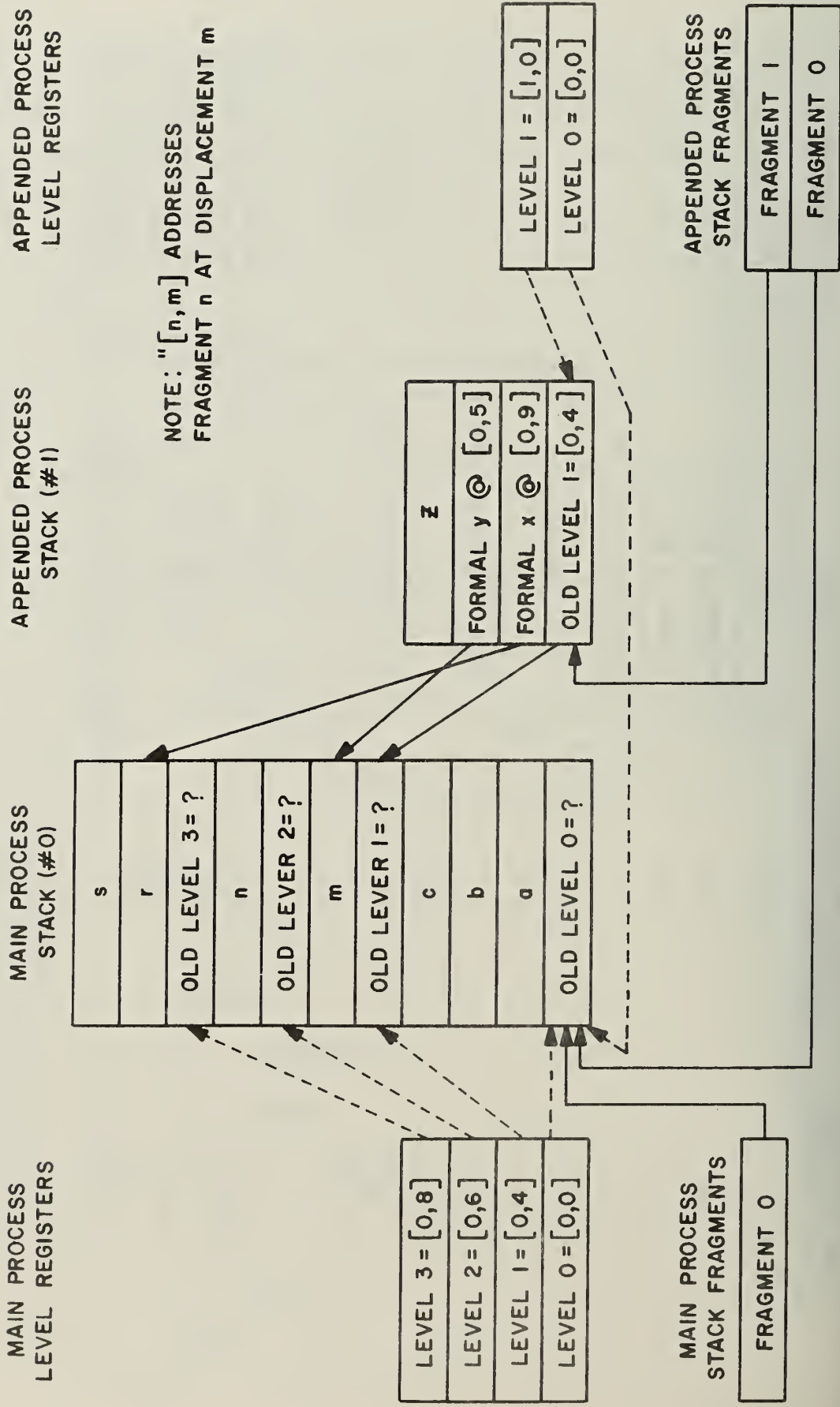


FIGURE 3. TWO STACK FRAGMENTS OF "GAL (mm)"

In this example the block levels have been numbered. The procedure "add1" creates variables at level 1, but it is passed variables from level 3 and level 1. It is not possible for a procedure running at level n to use level $n+1$ or higher level registers. Furthermore, it is not possible for the procedure "add1" to maintain two level 1 registers - one to address its own variable "Z" and one to address the actual parameter "m". To bypass these problems, base of stack relative addressing is used to pass the parameters.

By using base of stack relative addressing it is possible to create indirect reference words that can be used through any number of recursions. To enter the procedure in the above example an operator was executed to leave space to save level register 1. Then another operator was executed to create an indirect reference word to "r". This was repeated for "m". Finally the procedure entry operator was executed to relink the level register and enter a new code segment. So far this is very similar to the B6500 manner of addressing and procedure entry.

Things start to get sticky when procedures are appended as separate asynchronous processes rather than executed in line. In this case there must be several stacks (one for each appended process) each with the same base stack. If the procedure "add1" had been appended in the last example there would have been two processes running--each with its own set of level registers to define its environment.

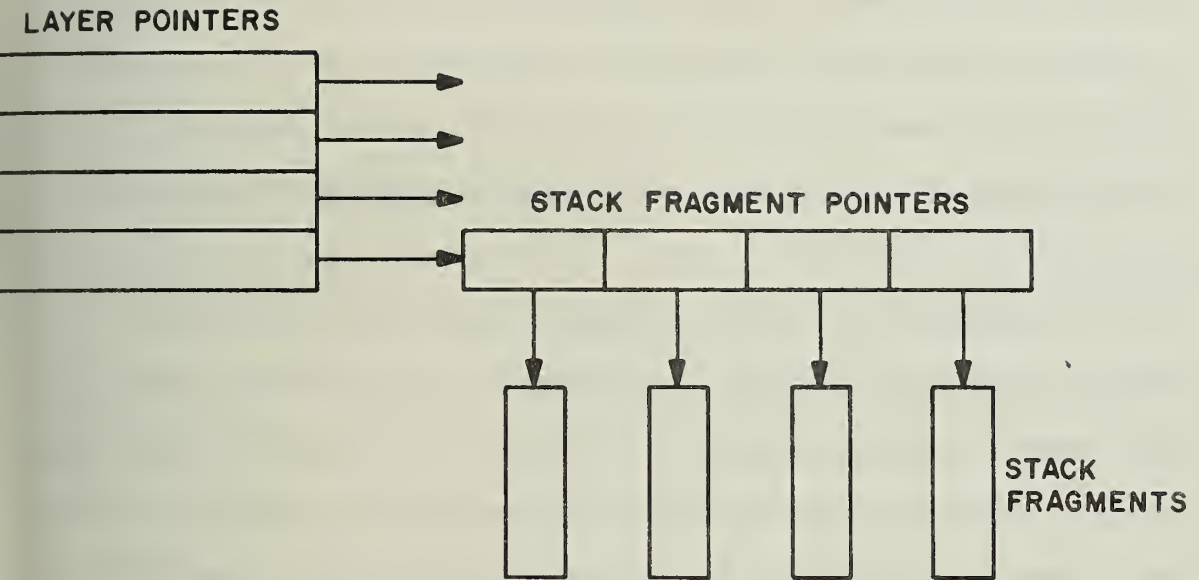
Note in Figure 3 that the level 0 register in the appended process points to level 0 in the main process. Therefore the main program and the appended procedure both reference the same level zero variables "a", "b", "c". However the level 1 registers in each process are different and reference different variables.

In addition to the array of level registers, an array of stack fragment registers is introduced. Now the operator that translates level displacement addresses into stack displacement indirect reference words not only produces a stack displacement but also produces a stack fragment index in the indirect reference word. If yet another "addl" were appended at this point, it would also receive separate arrays of level registers and stack fragment registers. The level 0 and stack 0 registers would point to the same place in both processes but each of the appended processes would have different level 1 and stack 1 pointers.

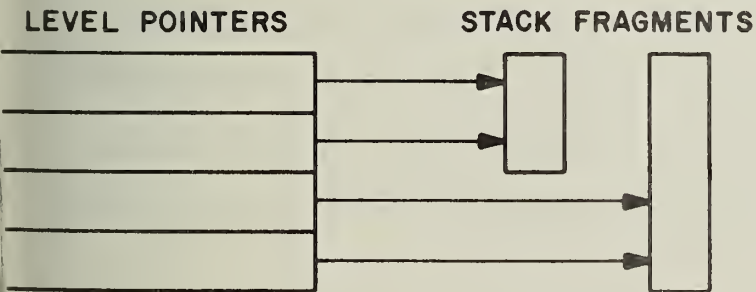
If a process now develops a level displacement address, it must point to a legitimate address for the process. There is no way a variable in another appended process or separate program can be addressed with a level displacement address.

Similarly stack displacement addresses used for formal parameters, cannot be created that point to illegal variables since they were all created at some time from a legitimate level displacement address. Clearly stack fragment 1 of the appended procedure "addl" provides no way of referencing a stack fragment of another appended "addl" or another program. The level registers and stack registers can be initialized using the environment checking techniques discussed earlier. Therefore the basic integrity of the values in the registers can be assured.

This is all fine for assuring a proper addressing space for a single multiprocess program or several running together. The problem now is to insure the integrity of the addressing space between operating system layers.



INDIRECT REFERENCE WORDS USE THE ADDRESS
TRIPLE (LAYER, STACK, DISPLACEMENT)



THE STANDARD ADDRESSING TECHNIQUE USES LEVEL DISPLACEMENT COUPLES.
THE LEVEL POINTER CONTAINS ALL REQUIRED STACK FRAGMENT REFERENCES.

Figure 4. Final addressing scheme.

To do this another array is introduced, the layer array. Furthermore, the processor is given the knowledge of the layer at which it is operating (This is just part of the initiate or restart operator.). When an indirect reference word is created, the operator also inserts the current system layer as well as the stack fragment and stack displacement into the word.

A complete set of layer, stack fragment, and level pointers defines the process environment (cf. Figure 4). Since higher layers of the operating system cannot create indirect reference words that point into lower layers, the higher layers have access only to the data in the lower layers that were passed to the higher layers as actual parameters. But to those variables they have direct access as if they were in their own layer. If one makes the final requirement that indirect reference words are tagged so that only special operators may access them, the integrity of the entire system is assured. Furthermore, foreign programs, appended processes, and layers are so arranged that no possible address can be created to reference them. Thus they are removed from the address space.

It should be noted that this scheme is not unreasonable. The idea of the stack fragment array, stack fragments, and level registers are used on the B6500. However the stack fragment array is a single array 102^4 long which contains every process in the machine. Thus the addressing spaces of programs are not disjoint. In order to address a second program a stuffed indirect reference word specifying its stack number would suffice. The OSL/2 concept makes this array part of the environment of each process thus separating the address spaces. The further addition of the layer array is straight forward.

The passing of information between layers is demonstrated to be feasible. The communication of semaphores as formal parameters is an obvious use of this interlayer communication. The problem of the addition and changing of primitives does not require esoteric hardware. Once interlayer addressability is defined, this problem falls under standard linkage editor techniques.

In conclusion, OSL/2 algorithmic techniques do not make severe requirements on the hardware architecture. However, the access variables imply a call by name facility which does require specialize hardware for efficient processing. The layering facilities do place severe restrictions on the hardware and require careful attention to the addressing problem.

Chapter 5

Conclusions

The data access techniques and process control concepts provide OSL/2 with the ability to create and manipulate queues and tables and to control multiple processes in a multilayered, hierarchial operating system. The procedure parameter specification facilities (keywords and defaults) are an integral part of these facilities and considerably enhance the readability of the language.

A system measurement capability is provided as a side effect of the data access facilities. The strong concepts of environment plus the measurement capability makes OSL/2 a language suitable for predictive system simulation as well as implemented system measurement.

OSL/2 implies a command driven operating system. That is, a system with multiple processes each either executing or else waiting for a command (P operation) to continue. There is no interrupt concept in OSL/2. In general, command driven systems (e.g., the "THE" system [10] and the ILLIAC IV system [11,12]) must have well defined structures in order to work in the command driven environment. Such systems are therefore more amenable to analysis, modification, and debugging.

The power of the general data access facilities is enormous. They allow the specification and manipulation of complex concepts as if they were primitive in the language. They further isolate the manipulation of special purpose constructs like queues and tables to a single declaration where access algorithms can be readily changed without touching any other portions of code. This facility specifies an access technique. The inter-

pretation of parameters and array information in the declaration and subscripts in statements and expressions are subject to programmer control. This allows a programmer to write his own semantics for variable accessing and control. Simple queues, complex tables, switch files, and even formatted I/O have been generated using this facility. It will take a significant amount of actual use before the range and power of this technique is well established.

The data access concepts can be added to existing languages like ALGOL, ESPOL, and PL/1. Furthermore, the concepts can be restricted and implemented in such a way that call by name hardware is not essential for their use. The structure definition in BLISS [22] is an example of a language with such a facility. However, adding such a capability to an assembly language is not reasonable. Assembly language does not have the richness of structure and concepts of scope that permit a clean implementation.

The layered system concept provided by OSL/2 is the most difficult feature to implement. Special hardware is certainly required. It is not recommended for implementation on current machines.

There may be one desirable concept left out of OSL/2. There is no primitive concept of a list in the syntax. Most lists are easy to build and maintain via the access techniques. However the concept of a written list is not available. This was important when writing formatted I/O. The desirable form of a read statement appeared to be read (<file>,<format string>,<list>) where <list> could be a list of identifiers, or expressions separated by commas. Instead of doing work this way, separate "read0", "read1", ..., "readn" procedures would be required to

read 0, 1, ..., n variables with "read" reserved for handling a list manipulated via the access facility. The author had used such a scheme in ALGOL 60 on the GE 635. In general, it was messy and one would continually change a "readk" to a "readn" to reflect a change in his list. Eventually, he gave up and went to the list form and declared many, many lists.

While the list is not incompatible with OSL/2, it was left out in an effort to keep an already large language from getting completely unmanageable. Lists can be added later if really needed. The author feels that data types like pattern also fall into this category. They were included only to document the compatibility of the SNOBOL pattern matching statement with an ALGOL-like block structured language.

Appendix A

A Reference to OSL/2 Syntax and Semantics

I. Introduction

OSL/2 is a block structured, Algol-based language designed for the specification of operating systems. OSL/2 is a follow-on language to OSL (Operating System Language) which was designed and documented by Alsberg and Wells [11].

OSL/2 is not designed for an existing machine, nor is it machine independent. Rather, OSL/2 is designed to make coding easy for the system's programmer, and it is also designed to be its own job control language. That is, a user in an OSL/2 system may use OSL/2 to control his job flow. To this extent OSL/2 is recursive in the system sense. One may describe new operating systems that exist on top of many layers of other operating systems.

It is expected that OSL/2 will be best implemented on a machine specifically designed to mirror the language itself. This machine will be called the OSL machine.

In the syntactic description of the language, the ALGOL 60 report is followed as closely as possible. Furthermore, every attempt is made not to arbitrarily introduce new terminology for concepts and entities available in, or similar to, other languages like ALGOL 60 and PL/1.

OSL differs sharply from ALGOL 60 in the following points. OSL includes list processing and generalized data access techniques. The programmer may specify special action to be taken when a fetch or store is performed on data declared with the general access facilities. These

facilities allow the ready specification of simple and complex queues, tables, and stacks.

Semaphores replace the notion of interrupts. Semaphores are used to synchronize processes.

Strings of various character sizes and varying lengths can be easily manipulated with facilities similar to those offered by SNOBOL.

In OSL/2 a process is any program or procedure that is capable of running asynchronously with any other process. Processes are created in one of three ways. Any currently running OSL process can "initiate" a new process. Initiated processes expect the initiating process to act as their operating system. The initiating process must explicitly indicate those procedure (primitives) within itself which the initiated process has access to. All other procedures and all data in the initiating process are not addressable by the initiated process. The initiating process may elect to default some or all existing operating system facilities to a lower level and choose only to add new facilities for the initiated process.

New processes may be "spawned" by an initiated process. This facility allows a running program to notify the operating system that another program (process) should be initiated by the operating system.

Finally, asynchronously running procedures (processes) may be "appended" to currently running processes. These appended procedures have access to all data and all procedures that they would have if they were executed sequentially. After appending a procedure, the appending process continues execution at the next executable statement and the appended process begins and continues execution at the same time.

II. OSL/2 Syntax and Semantics

Table of Contents

1. The Purpose and Extent of the Language
2. Basic Symbols, Identifiers, and Constants
 - 2.1. Letters
 - 2.2. Digits
 - 2.3. Constant Values
 - 2.4. Delimiters
 - 2.5. Special Symbols
 - 2.6. Identifiers
 - 2.7. Constants
3. Expressions
 - 3.1. Variables
 - 3.2. Function Designators
 - 3.3. Integer Expressions
 - 3.4. Boolean Expressions
 - 3.5. String Expressions
 - 3.6. Pointer Expressions
 - 3.7. Pattern Expressions
 - 3.8. Structure Expressions
4. Statements
 - 4.1. Compound Statements and Blocks
 - 4.2. Assignment Statements
 - 4.3. Dummy Statements

- 4.4. Case Statements
- 4.5. Procedure Statements
- 4.6. Process Statements
- 4.7. Iterative Statements
- 4.8. Escape Statements
- 4.9. Pattern Matching Statements
- 4.10. Synchronization Statements
- 4.11. Conditional Statements
- 5. Declarations
 - 5.1. Integer Declarations
 - 5.2. Boolean Declarations
 - 5.3. String Declarations
 - 5.4. Semaphore Declarations
 - 5.5. Process Declarations
 - 5.6. Pointer Declarations
 - 5.7. Structure Declarations
 - 5.8. Trap Declarations
 - 5.9. Pattern Declarations
 - 5.10. Access Definitions
 - 5.11. Access Declarations
 - 5.12. Procedure Declarations
 - 5.13. File Declarations
- 6. Compile Time Facilities
 - 6.1 Compile Time Declarations

1. The Purpose and Extent of the Language

OSL/2 is designed to describe the flow of control and manipulation of data required in an operating system. The term operating system refers primarily to supervisory and monitor systems. However, the string manipulation facilities make OSL/2 a powerful compiler writing language. OSL/2 is designed with the deliberate intent to make the generation of well structured and readable codes the natural coding mode. To this end the "GO TO" statement has been eliminated and the use of labels had been restricted to encourage self-documenting code.

A simple macro facility is specified.

The data base includes integers, Booleans, strings, semaphores, patterns, and pointers. Furthermore, structures, in the PL/1 sense, are available and extendible data access facilities have been provided.

2. Basic Symbols, Identifiers, and Constants

Basic Concepts.

The reference language is built up from the following basic symbols:

`<basic symbol> ::= <letter> | <digit> | <logical value> | <delimiter>
| <special symbol>`

2.1. Letters

`<letter> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|Ø|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|e
f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z`

This alphabet may be arbitrarily restricted, or extended with any other distinctive character (i.e., character not coinciding with any digit, logical value, or delimiter). Letters do not have individual meaning. They are used for forming constants and identifiers.

2.2. Digits

`<digit> ::= 0|1|2|3|4|5|6|7|8|9`

`<decimal digit> ::= <digit>`

`<binary digit> ::= 0|1`

`<quarternary digit> ::= 0|1|2|3`

`<octal digit> ::= 0|1|2|3|4|5|6|7`

`<hexadecimal digit> ::= 0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F`

Digits are used for forming integers, constants, and identifiers.

2.3. Constant Values

<logical value> ::= TRUE|FALSE

<string value> ::= MT

<pointer value> ::= NULL

Logical values are the values of Boolean quantities.

The string value MT denotes the empty string of length zero.

The pointer value NULL denotes the null pointer.

2.4. Delimiters

```

<delimiter> ::= <operator> | <separator> | <bracket> | <declarator>
               | <specifier>

<operator> ::= <integer operator> | <relational operator> | <pattern operator>
               | <logical operator> | <sequential operator> | <string operator>

<integer operator> ::= + | - | * | / | ↑ | RDIV | CDIV | MOD

<relational operator> ::= < | <= | = | >= | > | ≈ | LSS | LEQ | EQL
               | GEQ | GTR | NEQ

<logical operator> ::= EQV | IMP | ØR | AND | NØT | XØR | ~ | ANDIF | ØRIF

<pattern operator> ::= <vertical bar>

<sequential operator> ::= IF | THEN | ELSE | FI | DØ | SPAWN | APPEND
               | INITIATE | TERMINATE | SUSPEND | RESTART | STEP | TØ | BY
               | UNTIL | WHILE | FØR | P | V | LEAVE | CASE | ØF | ESAC

<string operator> ::= & | SAND | SØR | SNØT | SXØR

<separator> ::= , | . | : | ; | ← | := | CØMENT | TIMES | $
               | % | ! | <blank>

<blank> ::= <single space> | <blank> <single space>

<bracket> ::= ( | ) | [ | ] | BEGIN | END | ' | " | { | } | #

<declarator> ::= BØØLEAN | INTEGER | STRING | PØINTER | ACCESS
               | ØN | STRUCTURE | PRØCEDURE | NØDE | PRØCESS | FILE | PATTERN
               | QUEUE | STACK | ØWN | DEFINE | PRIMITIVE | SEMAPHØRE

<specifier> ::= VALUE | NAME | FIXED | FØRMAL | FØRWARD

<vertical bar> ::= |

```

Delimiters separate the various entities of the language. In order to accept input from common restricted alphabets, the following symbols and combinations of symbols are equivalent:

<u>Standard Symbols</u>	<u>Alternatives</u>
<	LSS
<=	LEQ
=	EQL
>=	GEQ
>	GTR
≠	NEQ
←	:=
~	NOT

what follows, only the left hand standard symbols are used.

Delimiters have fixed meanings which will be explained as they occur in various constructs. Delimiters and logical values are considered basic symbols of the language and have no relation to the individual letters of which they are composed. Therefore, the words which constitute basic symbols are reserved for specific use in the language.

4.1. Spacing

No space may appear between the letters of a reserved word. At least one space must separate a multicharacter delimiter from any adjacent letter or digit. That is, any two basic components of the following form must be separated by <blank>.

1. Multicharacter delimiter
2. Identifier
3. Logical value
4. String value
5. Pointer value
6. Unsigned integer

2.4.2. Comments

Comments may be inserted in the program without effect on the program structure. The following conventions hold:

- 1) {terminal symbol}COMMENT {any sequence of symbols except;} ;
is equivalent to {terminal symbol}
- 2) % {any sequence of symbols except %} % is equivalent to
<blank>
- 3) ! {any sequence of symbols} {end of line} is equivalent
to <blank>
- 4) END {any sequence of letters digits and blanks} is equivalent to END

The % convention has precedence over all other comment conventions.

No comment conventions are recognized inside strings.

The metalinguistic notation {...} is used to denote the terminal symbol or group of terminal symbols indicated by the english phrase enclosed in the metalinguistic braces.

2.5. Special Symbols

<special symbol> ::= _

The underscore is completely ignored in the language except when it appears within a string. The underscore is used in identifiers and constants to improve readability.

2.6. Identifiers

2.6.1. Syntax

`<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>`

2.6.2. Examples

A1

PRØCESSØR_NUMBER

TIME_LIMIT

JØB23

X

2.6.3. Semantics

No spaces may appear within identifiers. The underscore "_" is used to improve readability, but is not part of the identifier. Thus "A_B", "AB", and "A__B" are all the same identifier.

Identifiers are used to identify pointers, events, procedures, variables, etc.

Reserved words may not be used as identifiers. Otherwise, identifiers may be chosen freely.

Identifiers may be of any length. However, finite implementations are envisioned. In no case will the maximum identifier size be less than fifteen characters. When reduction of an oversized identifier is required, the identifier shall be reduced by the following algorithm:

- 1) n is the maximum identifier length
- 2) if n is even, choose $n/2$ characters from the front and $n/2$ characters from the rear of the identifier
- 3) if n is odd, choose $(n+1)/2$ characters from the front and $(n-1)/2$ characters from the rear of the identifier.

2.7. Constants

2.7.1. Integers

2.7.1.1. Syntax

$\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle$

$\langle \text{integer} \rangle ::= \langle \text{unsigned integer} \rangle \mid + \langle \text{unsigned integer} \rangle \mid - \langle \text{unsigned integer} \rangle$

2.7.1.2. Examples

1

123_456_789_012

+17

-35_702

2.7.1.3. Semantics

The underscore is used to improve readability and does not alter the value of the integer. The value of the integer is its decimal based value.

2.7.2. Strings

2.7.2.1. Syntax

$\langle \text{binary string} \rangle ::= 2' \langle \text{binary digit string} \rangle'$

$\mid 2'' \langle \text{binary digit string} \rangle''$

$\langle \text{quaternary string} \rangle ::= 4' \langle \text{quaternary digit string} \rangle'$

$\mid 4'' \langle \text{quaternary digit string} \rangle''$

$\langle \text{octal string} \rangle ::= 8' \langle \text{octal digit string} \rangle'$

$\mid 8'' \langle \text{octal digit string} \rangle''$

$\langle \text{decimal string} \rangle ::= 10' \langle \text{unsigned integer} \rangle'$

$\mid 10'' \langle \text{unsigned integer} \rangle''$

$\langle \text{hexadecimal string} \rangle ::= 16' \langle \text{hexadecimal digit string} \rangle'$

$\mid 16'' \langle \text{hexadecimal digit string} \rangle''$

<binary digit string> ::= <binary digit>

| <binary digit string> <binary digit>

<quaternary digit string> ::= <quaternary digit>

| <quaternary digit string> <quaternary digit>

<octal digit string> ::= <octal digit>

| <octal digit string> <octal digit>

<hexadecimal digit string> ::= <hexadecimal digit>

| <hexadecimal digit string> <hexadecimal digit>

<character string> ::= <symbol string> | A <symbol string>

| E <symbol string> | B <symbol string>

<symbol string> ::= "{any string of symbols}" | '{any string of symbols}'

<string> ::= <binary string> | <quaternary string> | <octal string>

| <decimal string> | <hexadecimal string> | <character string> | M

Each quote, identical to the opening quote, which is placed within the string, must be presented by two juxtaposed quotes of the same type as the opening quote.

2.7.2.2. Examples

2'011001101'

16'FAEC0628F'

8'73107777'

"NØW'S THE TIME"

'WHAT''S YØUR PRØBLEM?'

'TØ QUØTE: "GØ HØME!"'

E" (\$ # @ ""?"

2.7.2.3. Semantics

The number strings described are used as masks and can take the positive value of their binary representation when used in arithmetic

expressions. A decimal string is semantically equivalent to a hexadecimal string.

MT is the universal null string -- it is empty.

Character strings are, by default, eight bit ASCII code. The prefix A denotes seven bit ASCII, E denotes eight bit EBCDIC, and B denotes 6 bit IBM BCD. Either double or single quotes may open a string. A string is closed by the same type of quote which opened it.

3. Expressions

In the language, the primary constituents of the program which describe algorithmic processes are integer, Boolean, string, pointer, pattern, and structure expressions.

$$\begin{aligned} \langle \text{expression} \rangle ::= & \langle \text{integer expression} \rangle \mid \langle \text{Boolean expression} \rangle \\ & \mid \langle \text{string expression} \rangle \mid \langle \text{pointer expression} \rangle \\ & \mid \langle \text{structure expression} \rangle \mid \langle \text{pattern expression} \rangle \end{aligned}$$

3.1. Variables

3.1.1. Syntax

```

<subscript list> ::= <subscript expression>
                    | <subscript list>,<subscript expression>
<subscript expression> ::= <integer expression> | <actual parameter>
<substring specification> ::= [<first position>:<number of positions>]
                    | [<first position>:<number of positions>:<size>]
<first position> ::= <integer expression>
<number of positions> ::= <integer expression>
<size> ::= <integer expression>
<name> ::= <identifier> | <structure first part>.<identifier>
          | <identifier> [<subscript list>]
          | <structure first part>.<identifier> [<subscript list>]
<structure first part> ::= <structure name>
          | <structure first part>.<structure name>
<structure name> ::= <identifier> | <identifier> [<subscript list>]
<pointer name> ::= <name>
<pointer specifier> ::= <pointer name>.|<pointer-specifier> <pointer name>.
<variable> ::= <name> | <pointer specifier> <name>
<string variable> ::= <variable> | <variable>.<substring specification>
<structure variable> ::= <structure first part>
          | <pointer specifier> <structure first part>

```

3.1.2. Examples

A

A[23]

USER_NAME.[1:20]

STUDENT_ENTRY[N*2+K].ID_NØ

P.LIST_ELEMENT

X.[27:3]

3.1.3. Semantics

Variables name data quantities. A variable may specify single quantities, multidimensional arrays, or whole data structures.

When a variable specifies an array, each subscription expression is evaluated and used to determine which array element is specified. The value of a subscripted variable, one or more of whose subscripts are out of bounds, is undefined. If an out of range subscript is calculated, an "index" trap occurs. Access variables may have actual parameters passed through the subscript list. In this case, the interpretation of the subscripts is given by the associated access definition.

A variable which specifies a string quantity, may be followed by a substring specification. The substring specification indicates what is the first character desired in the string and how many contiguous characters to the right of the first character (including the first character) are desired. The character positions in a string are numbered from left to right beginning at zero. If a size parameter is specified, the new character size is used instead of what was declared for the string.

The symbol "." is used in variables to exactly define which quantity is desired when a pointer is specified. If P is a pointer, then "P" denotes the pointer P and "P." denotes the quantity pointed to by P. This use of the "." is consistent with structure addressing.

Each higher level identifier used in a structure, that is each identifier that does not identify a primitive data item, may be viewed as a pointer, e.g.

```
STRUCTURE JØB (INTEGER ID, TIMEØN, TIMEØFF;
                STRING NAME (30);
                STRUCTURE IDCARD_PARDS(INTEGER TIME,LINES));
```

"JØB" and "IDCARD_PARDS" are higher level identifiers in this structure.

Thus one would address "LINES" as follows:

```
JØB.IDCARD_PARDS.LINES
```

If "JØB" were a structure which was repeated many times in a list and "P" were a pointer which pointed into this list then the "JØB" structure pointed to by "P" would be expressed as "P.JØB".

3.2. Function Designators

3.2.1. Syntax

`<procedure identifier> ::= <identifier>`

`<actual parameter> ::= <keyword parameter> | <parameter>`

`<keyword parameter> ::= <formal parameter identifier> : <parameter>`

`<parameter> ::= <expression> | <procedure identifier>`

`| <variable> | * | <empty>`

`<actual parameter list> ::= <actual parameter>`

`| <actual parameter list>, <actual parameter>`

`<function designator> ::= <procedure identifier>`

`| <procedure identifier> (<actual parameter list>)`

The variables which appear as actual parameters may be semaphores, files, access variables, or process variables.

3.2.2. Examples

`GETSPACE (100*JØB_SIZE)`

`ELAPSED_TIME (A,B,C,D)`

`GLØRPSHNØTT (MERKLE_JAMMER)`

3.2.3. Semantics

Function designators may define any data type value available in OSL/2. This data is returned by the function after it executes a given set of rules defined by a procedure declaration.

3.2.4. Standard Functions

The following functions (primitives) are predefined and implemented for the user.

<u>Functions</u>	<u>Arguments</u>	<u>Action</u>
ABS(E)	integer expression	absolute value of E is returned.
SIGN(E)	integer expression	if E>0 return +1, if E=0 return 0, if E<0 return -1.
DECIMAL(E)	integer expression	returns a decimal string which represents the decimal value of ABS(E).
BINARY(E)	decimal string	returns the binary integer value of the string E.
CONVERT(A,B,C) or CONVERT(A,B,C,N)	strings A,B,C and expression N	convert N characters in string A into a new string using truncated values of the characters in string C. Each character in A. is used to index into string C. The first character in C is selected if the character in A has value 0, the second if it has value 1, etc. The Character in C is truncated on the left or padded with zeroes on the left to match the character size of B. The selected character is then placed in B and the process continues at the next position in A and B until N characters have been processed. If the character position beyond the end of the string C, or if N exceeds the size of A or B then a convert failure trap is caused. If $N \leq 0$ then no characters are converted.
LENGTH(E)	string	returns the number of characters in string E.
CHARSIZE(E)	string	returns the number of bits per character in string E.

<u>Functions</u>	<u>Arguments</u>	<u>Action</u>
PTR(E)	any single variable or array element or character	A pointer to the variable, array element, or character is returned.
SPAN(E) or SPAN(E,N)	E=string N=integer expression	A pattern is returned which represents a string of length N consisting of any of the characters in E. If N is absent, a string of arbitrary length is matched.
EXCEPT(E) or EXCEPT(E,N)	E=string N=integer expression	Same as SPAN except that a string which consists of any characters except those in E is matched.
ANY or ANY(N)	N=integer expression	ANY is a pattern which matches any string. If N is specified, then ANY matches any string of length N.
RELEASE(E)	variable	The specified variable is deallocated.
UPPER(E)	array row specification	The upper bound of the speci- fied array row is returned.
LOWER(E)	array row specification	The lower bound of the speci- fied array row is returned.
STRING(E) or STRING(E,N)	integer expression	E is converted to a bit string of length N. Two's complement arithmetic is assumed. If N not provided, the current default length of an integer is assumed.
ALLOCATE(E)	identifier	A pointer is returned with the location of E in it. This primitive is used at block entry and block exit. If it is redefined by the user, the user may do his own allocation. However, the user may only allocate space in arrays declared lexico- graphically prior to the redefinition of primitive ALLOCATE. This function may also be used as a procedure statement.

<u>Functions</u>	<u>Arguments</u>	<u>Action</u>
BØØL(E)	integer expression	Allows E to be used as a Boolean with Boolean operators. BØØL(E) is true if E is odd or has a right-most bit equal to 1. Otherwise it is false.
INTGR(E)	decimal string	Allows E to be used as an integer.

3.3. Integer Expressions

3.3.1. Syntax

<unary operator> ::= + | -

<adding operator> ::= + | -

<multiplying operator> ::= * | / | RDIV | CDIV | MOD

<primary> ::= <unsigned integer> | <variable> | <function designator>
 | CASE <integer expression> OF <integer expression list> ESAC
 | (<integer expression>)
 | IF <Boolean expression> THEN <integer expression> ELSE
 <integer expression> FI

<factor> ::= <primary> | <unary operator> <primary> | <factor> ↑ <primary>
 | <factor> ↑ <unary operator> <primary>

<term> ::= <factor> | <term> <multiplying operator> <factor>

<simple integer expression> ::= <term>
 | <simple integer expression> <adding operator> <term>

<integer expression> ::= <simple integer expression> | <integer assignment>

<integer expression list> ::= <integer expression>
 | <integer expression list>, <integer expression>

3.3.2. Examples

Integer expressions:

A+B+C

X↑-3

-4 * (A+INTGR (16'OAFF') MOD 3)

18+ IF JØBTIME MAXTIME> THEN A/B

ELSE IF A * B MOD C = X

THEN RØUNDER (Q)

ELSE A/B FI FI

$-(Q \leftarrow B+C)$

$A + -G * 30$

$Q + \text{INTGR} (X.[17:5])$

$X \leftarrow A+B * (C \leftarrow 10*L)$

Terms:

$Z [17, A * B] * \text{TEST\_N}\emptyset$

$A * B * X [11, \text{IF } A = B \text{ THEN } 3 \text{ ELSE } 4 \text{ FI}] * Q$

Factors:

$-A$

B

$C \uparrow D \uparrow -E \uparrow 10$

$+(30 * A)$

$-(A \uparrow (10+J))$

Primaries:

73

$\text{IF } Q = R \text{ ANDIF } B\emptyset\emptyset L (X + 1) \text{ IMP } B \text{ THEN } 17 \text{ ELSE } Q \text{ FI}$

$\text{LENGTH} (P)$

$\text{BASE} (\text{PTR1})$

$(X \leftarrow \text{IF } P \text{ THEN } 2 \text{ ELSE } C * Q \text{ FI})$

3.3.3. Semantics

An integer expression is a rule for computing a numerical value. This value is obtained by executing the indicated arithmetic operations on the actual numerical value of a primary. The value is obvious in the case of constants. For variables, it is the current value (assigned last in the dynamic sense), and for function designators it is the value returned when the computing rules defining the proce-

dure are applied to the current values of the procedure parameters given in the expression. For integer expressions enclosed in parentheses, the value must, through a recursive analysis, be expressed in terms of values of primaries of the other three kinds.

The value of a primary of the form IF <Boolean expression> THEN <integer expression 1> ELSE <integer expression 2> FI is computed as follows. The Boolean expression is evaluated. If it is true, integer expression 1 is evaluated and is the value of the primary. If the value of the Boolean expression is false, then integer expression 2 is evaluated and is the value of the primary.

The value of a primary of the form CASE <integer expression> OF <integer expression₀>, <integer expression₁>, ... <integer expression_n> ESAC is computed as follows. The integer expression following the case is evaluated. Assume it has value i . If $0 \leq i \leq n$ then integer expression _{i} is calculated and is the value of the primary. If $i < 0$ or $n < i$, then integer expression _{n} is calculated and is the value of the primary.

If an assignment statement appears as an arithmetic expression, the arithmetic assignment statement is executed and the value of the expression is the value assigned by the assignment statement.

If side effects in a function designator will cause ambiguity of result, expressions are evaluated in the order specified by the syntax.

3.3.3.1. Operators and Types

The values used in evaluating integer expressions are always integers. The resulting value of the integer expression is an integer.

3.3.3.1.1. The operators $+$, $-$, and $*$ have the conventional meanings of addition, subtraction and multiplication. $\uparrow$ denotes exponentiation.

3.3.3.1.2. The operators $/$, RDIV, CDIV, and MOD have special meanings.

Let $\div$ denote normal division of arbitrary real numbers, and let $\text{Entier}(x)$ denote the largest integer less than or equal to x . Then

A/B is a lower value divide $= \text{Entier}(A \div B)$

$A \text{ RDIV } B$ is a rounded divide $= \text{Entier}(A \div B + .5)$

$A \text{ CDIV } B$ is a ceiling value divide $= -\text{Entier}(-(A \div B))$

$A \text{ MOD } B$ is the remainder after dividing A by $B = (A - (A/B) * B)$

If A or B are expressions, then they are called by value before executing the above algorithms.

3.3.3.1.3. The levels of precedence as defined by the syntax are as follows:

first: unary $+$, $-$

second: $\uparrow$

third: $*$, $/$, RDIV, CDIV, MOD

fourth: binary $+$, $-$

3.4. Boolean Expressions

3.4.1. Syntax

<relational operator> ::= < | <= | = | >= | > | ~ =

<equality operator> ::= = | ~ =

<relation> ::= <integer expression> <relational operator> <integer expression>

| <string expression> <relational operator> <string expression>

| <pointer expression> <equality operator> <pointer expression>

<Boolean primary> ::= <logical value> | <variable>

| <function designator> | <relation> | (<Boolean expression>)

| IF <Boolean expression> THEN <Boolean expression> ELSE

<Boolean expression> FI

| CASE <integer expression> ØF <Boolean expression list> ESAC

<Boolean secondary> ::= <Boolean primary> | <Boolean primary>

<Boolean factor> ::= <Boolean secondary>

| <Boolean factor> AND <Boolean secondary>

| <Boolean factor> ANDIF <Boolean secondary>

<Boolean term> ::= <Boolean factor> | <Boolean term> XØR <Boolean factor>

| <Boolean term> ØR <Boolean factor>

| <Boolean term> ØRIF <Boolean factor>

<implication> ::= <Boolean term> | <implication> IMP <Boolean term>

<simple Boolean> ::= <implication> | <simple Boolean> EQV <implication>

<Boolean expression> ::= <simple Boolean> | <pattern matching statement>

| <Boolean assignment>

<Boolean expression list> ::= <Boolean expression>

| <Boolean expression list>, <Boolean expression>

Variables and function designators used as Boolean primaries must be declared of type BØØLEAN.

3.4.2. Examples

A = B

"HI THERE!" = ACKNOWLEDGEMENT

IF SOME_BOOLEAN THEN TRUE ELSE F(G * T)FI

IF IF A = B THEN Q ELSE R(P3)FI THEN TRUE ELSE X = R FI

(A = B XOR C) AND D EQV L IMP Z23

BOOLEAN ← INT1 = INT2 ← 2*INT3

3.4.3. Semantics

A Boolean expression is a rule for computing a logical value. The rules of evaluation are completely analogous to rules for evaluating integer expressions.

Relations take the value true if the relation is satisfied for the expressions involved. If the relation is not satisfied, then the value of the relation is false. Only strings of identical character size may be used in a relation. When strings are being compared, the comparison takes place from left to right character by character until an inequality is determined or the two strings prove to be equal in content and length. Characters are compared by the arithmetic value of their binary representations. If two strings of unequal length are compared and the entire shorter string is identical to the first part of the longer string, then the shorter string is by definition less than the longer string. Two pointers are equal if they point to the same data.

3.4.3.1. Operators

The meaning of the logical operators is defined in the following table:

OPERANDS

OPERATIONS

b1	b2	¬b1	b1 AND b2	b1 XOR b2	b1 ØR b2	b1 IMP b2	b1 EQV b2
false	false	true	false	false	false	true	true
false	true	true	false	true	true	true	false
true	false	false	false	true	true	false	false
true	true	false	true	false	true	true	true

The sequence of operations within one expression is left to right with the following rules of precedence:

- first: integer, string, and pointer expressions
- second: <, <=, =, >=, >, ¬=
- third: ¬
- fourth: AND, ANDIF
- fifth: XOR, ØR, ØRIF
- sixth: IMP
- seventh: EQV

The operators ANDIF and ØRIF have the same meaning as AND and ØR except that they may alter the evaluation rules of the Boolean expression and may thereby drastically change any side effects. If the logical value computed to the left of an ANDIF in the current Boolean factor is false, then no further calculation is made to the right in the current Boolean factor containing the ANDIF. The value of the Boolean factor is false. If the logical value computed to the left of an ØRIF in the current Boolean term is true, then no further calculation is made to the right in the current Boolean term containing the ØRIF. The value of the Boolean term is true. If the above conditions do not hold, the action is identical to AND and ØR.

Strings and integers may be used as Booleans when indicated by the `BOOL` function. In this case the string is treated as true if the right-most bit is 1 and false if it is 0. An integer becomes true if it is odd and false if it is even.

3.5. String Expressions

3.5.1. Syntax

```

<string primary> ::= <string constant> | <function designator> | <variable>
                    | (<string expression>)
                    | IF <Boolean expression> THEN <string expression>
                      ELSE <string expression> FI
                    | CASE <integer expression> OF <string expression list> ESAC
<string secondary> ::= <string primary> | SNØT <string primary>
<string factor> ::= <string secondary> | <string factor> SAND <string secondary>
<string term> ::= <string factor> | <string term> SØR <string factor>
                 | <string term> SXØR <string factor>
<simple string expression> ::= <string term>
                             | <simple string expression> & <string term>
<string expression> ::= <simple string expression> | <string assignment>
<string expression list> ::= <string expression>
                             | <string expression list>, <string expression>

```

Variables and functions are used as string primaries must be declared to be type STRING. Only strings of the same character size may be concatenated.

3.5.2. Examples

```

"THIS" & "IS A CØNCATENATED STRING"

DECIMAL (SALARY) & (IF CENTS THEN 10"00" ELSE MT FI)

A SAND 2'110000111100'

```

3.5.3. Semantics

A string expression is a rule for computing a string value. The rules for evaluating a string expression are analogous to the rules for evaluating an integer expression.

The string operators SAND, SØR, SXØR, and SNØT correspond respectively to bit wise and, or, exclusive or, and not operations. These operators work left to right on any strings. The operators SAND, SØR, and SXØR may operate on strings of unequal length. The result is the same as if the operations were performed on the full length of the shorter operand and the first part of the longer operand (which is the same length as the shorter operand) concatenated with the rest of the longer operand.

The string constant MT is the empty string.

The value of a string expression is the concatenated string whose length is equal to the sum of its concatenated components.

3.6. Pointer Expressions

3.6.1. Syntax

`<pointer expression> ::= <pointer variable>`

`| <function designator> | NULL`

`| IF <Boolean expression> THEN <pointer expression>`

`ELSE <pointer expression> FI`

`| CASE <integer expression> OF <pointer expression list> ESAC`

`| <pointer assignment>`

`<pointer expression list> ::= <pointer expression>`

`| <pointer expression list>, <pointer expression>`

The pointer variables and function designator in pointer expression must be of type POINTER.

3.6.2. Examples

`P.A.PTR1`

`P.A`

`P ← PTR(B)`

`IF R1 = P2 THEN P1 ELSE P1.P FI`

`P[23]`

`P[A*B+C]`

3.6.3. Semantics

A pointer expression is a rule for computing a pointer value. A pointer value points to a particular piece of data whether it be an individual character, integer, array, event, structure, etc. The pointer operator "." may be read right to left as "pointed to by the pointer". Thus "P.A" is A pointed to by the pointer P.

3.7. Pattern Expressions

3.7.1. Syntax

```

<pattern primary> ::= <string> | <variable> | <function designator>
    | <pattern expression>
    | IF <Boolean expression> THEN <pattern expression>
      ELSE <pattern expression> FI
    | CASE <integer expression> OF <pattern expression list> ESAC

```

```

<pattern secondary> ::= <pattern primary>

```

```

    | <pattern secondary> & <pattern primary>

```

```

<pattern factor> ::= <pattern secondary> | <string variable>

```

```

    | <pattern secondary>

```

```

<pattern alternative list> ::= <pattern factor>

```

```

    | <pattern alternative list> <vertical bar> <pattern factor>

```

```

<pattern expression> ::= {<pattern alternative list>}

```

```

<pattern expression list> ::= <pattern expression>

```

```

    | <pattern expression list>, <pattern expression>

```

3.7.2. Examples

```

"A"

```

```

{B ← {"A" | "B" & SPAN("012376")}} | B ← X & C}

```

```

{"LMQ" | "CAT" | "DOG"}

```

3.7.3. Semantics

Pattern expressions define a rule for scanning strings. Patterns are used in string matching statements and their semantic interpretation is discussed there.

3.8. Structure Expressions

3.8.1. Syntax

```

<structure expression> ::= <structure variable>
    | <function designator>
    | IF <Boolean expression> THEN <structure expression>
      ELSE <structure expression> FI
    | CASE <integer expression> OF <structure expression list> ESAC
<structure expression list> ::= <structure expression>
    | <structure expression list>, <structure expression>

```

Variables and functions used as structure expressions must be declared to be structures. All structures in one expression must have the same hierarchy and types in each position.

3.8.2. Examples

A

A.B

IF A=B THEN S ELSE B.S FI

3.8.3. Semantics

The rules for evaluating a structure expression are analogous to the rules for evaluating an integer expression.

4. Statements

The basic unit of calculation is the statement. Statements are normally executed sequentially in the order in which they are written. This sequence may be shortened by conditional or escape statements which may cause some statements to be skipped, or process statements may create new processes that will run in parallel with the current code.

Statements may be grouped into compound statements and blocks which are themselves statements. Therefore the definition of statement is necessarily recursive.

4.1. Compound Statements and Blocks

4.1.1. Syntax

```

<basic statement> ::= <assignment statement> | <dummy statement>
                    | <case statement> | <procedure statement> | <process statement>
                    | <iterative statement> | <escape statement>
                    | <pattern matching statement> | <synchronization statement>

<unconditional statement> ::= <basic statement> | <compound statement>
                             | <block>

<statement> ::= <unconditional statement> | <conditional statement>

<statement list> ::= <statement> | <statement list>;<statement>

<declaration list> ::= <declaration> | <declaration list>;<declaration>

<compound statement> ::= BEGIN <statement list> END
                       | <label> : <compound statement> : <label>

<block head> ::= BEGIN <declaration list>

<block> ::= <block head>;<statement list> END
          | <label> : <block> : <label>

<program> ::= <block> | <compound statement> | <procedure declaration>

```

All labels are used as brackets. If a label is used before a statement, the same label must be used after the statement.

4.1.2. Examples

Let S, D, and L denote arbitrary statements, declarations and labels.

Basic statements:

form: S

A $\leftarrow$ B+C

P.D $\leftarrow$ Q

LEAVE OUTERBLOCK

Compound statements:

form: L:L...L:BEGIN S;S;S;...;S END:L ... L:L

CD:BEGIN

P(23,17,A*B);

X $\leftarrow$ P+D;

REWIND(TAPE9);

TERMINATE

END:CD

Blocks:

form: L:L...L:BEGIN D;D;...;D;S;...;S END : L:L ...L

BLOCK: BEGIN

INTEGER X,Y,Z;

STRING (A,B,C) (30);

A $\leftarrow$ "NOW IS THE TIME";

B $\leftarrow$ A & "23 SKIDØØ";

$X \leftarrow 2*(Y \leftarrow 30);$

$Z \leftarrow X+Y;$

IF $X = \text{INTGR}(10''37'')$ THEN LEAVE BLOCK ELSE TERMINATE FI

END: BLOCK

4.1.3. Semantics

Every block introduces a new level of nomenclature. Any identifier occurring within the block is local to the block if it is defined in the blockhead. Identifiers defined outside the present block but within a block containing the present block are global to the present block. Only entities represented by local and global identifiers have any existence within a block. When an identifier is declared within a block, any entity represented by this identifier outside the block is completely inaccessible inside the block.

4.2. Assignment Statements

4.2.1. Syntax

```
<assignment statement> ::= <integer assignment> | <Boolean assignment>
                        | <string assignment> | <pointer assignment>
                        | <pattern assignment> | <structure assignment>
```

```
<left part list> ::= <variable> ← | <left part list> <variable> ←
```

```
<integer assignment> ::= <variable> ← <integer expression>
```

```
<Boolean assignment> ::= <variable> ← <Boolean expression>
```

```
<string assignment> ::= <variable> ← <string expression>
```

```
<pointer assignment> ::= <left part list> <pointer expression>
```

```
<structure assignment> ::= <left part list> <structure expression>
```

```
<pattern assignment> ::= <left part list> <pattern expression>
```

All variables on the left of the assignment arrow must be the same type as the expression on the right. An asterisk may be used as the first primary in the expression on the right of the assignment arrow. Any pointer or pattern variable appearing on the left of an assignment arrow must be at the same or lower nested block level of any variables appearing to the right of the arrow. If the variable on the left of an assignment is a function identifier, then the expression value is saved to be returned at the end of the procedure.

4.2.2. Examples

```
A ← B+C
```

```
X ← *+1
```

```
B ← X = Y
```

```
S ← "TEST"
```

```
P ← PTR(A[0])
```

```
S1 ← S2
```

```
P ← {"A" | "B" | "CAT"}
```

4.2.3. Semantics

Assignment statements assign the value of an expression to one or more variables in a left part list. The types of the entities on the left and right of the assignment must agree. No type transfers are automatically invoked.

An asterisk in place of the first primary after an assignment arrow means that the variable to the immediate left of the assignment arrow is to be used in place of the asterisk without further evaluation of subscripts, etc. that may cause side effects. An access variable may not be this variable if the asterisk convention is used.

The presence of an access identifier on the left or right of an assignment automatically invokes the appropriate fetch or store procedure specified in the access declaration to generate or save an expression value.

Assignment statements are executed in the order specified by the following algorithm:

- first: Any subscript or substring specifications occurring in the left part list are evaluated in sequence from left to right.
- second: The expression of the statement is evaluated.
- third: The value of the expression is assigned to all the left part variables, and all access variable stores are performed from right to left.

4.3. Dummy Statements

4.3.1. Syntax

<dummy statement> ::= <empty>

4.3.2. Examples

BEGIN...; END

...;;...

4.3.3. Semantics

A dummy statement executes no rules and performs no calculations. It is completely null. Dummy statements may serve as markers in unused indices of a case statement.

4.4. Case Statements

4.4.1. Syntax

```
<case statement> ::= CASE <integer expression> ØF <statement list> ESAC
    | <label> : <case statement> : <label>
```

4.4.2. Examples

```
CASE N ØF
```

```
    P0(A,B);
```

```
    P1(A,B,);
```

```
    P2(A,B)
```

```
ESAC
```

```
RESET: CASE GLØRP ØF
```

```
    PX ← PTR (A[K]);
```

```
    PX ← PTR (B[K]);
```

```
    PX ← PTR (C[K])
```

```
ESAC: RESET
```

4.4.3. Semantics

Case statements allow one statement to be selectively executed from a list of statements. The statements in the statement list are numbered 0,1,2,...,n. The integer expression is evaluated. If the value of the arithmetic expression is k then the kth statement is executed. If k is out of range - i.e., k < 0 or k > n then the nth statement is executed. Control proceeds to the statement immediately following the ESAC after the appropriate statement is executed.

4.5. Procedure Statements

4.5.1. Syntax

```
<procedure statement> ::= <procedure identifier>
    | <procedure identifier> (<actual parameter list>)
```

4.5.2. Example

```
SIN(X)
```

```
INNER_PRODUCT(A,B)
```

4.5.3. Semantics

A procedure statement serves to invoke (call for) the execution of a procedure body (cf. procedure declarations). The effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement.

4.5.3.1. Call by Value

All formal parameters quoted in the value part of the procedure declaration heading are assigned the values of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body in the order in which the parameter appears in the formal parameter list. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variable local to this fictitious block with types as given in the corresponding specifications. As a consequence, variables called by value are to be considered as nonlocal to the body of the procedure, but local to the fictitious block.

4.5.3.2. Call by Name

Any formal parameter quoted in the name list is replaced, throughout the procedure body, by the corresponding actual parameter after enclosing

this latter in parenthesis wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

4.5.3.3. Call by Reference

Any formal parameter not quoted in the name or value list is called by reference. References which consist of a single variable, array, structure, substructure, or element of an array are called by name after having all subscripts of such variables called by value. Any pattern variables and structure variables containing patterns and all other variables and expressions will be called by value.

4.5.3.4. Body Replacement and Execution

The procedure body, modified as above, is inserted in place of the procedure statement and executed. If the procedure is called from a place outside the scope of any nonlocal quantity of the procedure body the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

4.5.4. Syntactic Restrictions

4.5.4.1. Actual-Formal Correspondence

Actual parameters must normally correspond in number and type to formal parameters in the procedure declaration in the order in which they appear in both the actual and formal parameter lists unless the actual parameters are keyword parameters (cf. §3.2.1. & §4.5.4.2.) or default parameters (cf. §5.12.3. & §4.5.4.3.). Furthermore, subscript ranges, string lengths, and bases must agree with the actual parameters as they are defined in the procedure declaration heading.

4.5.4.2. Keyword Specifications

Actual parameters which are preceded by a keyword (formal parameter identifier followed by a colon) take one position in the ordered list but are linked to the indicated formal parameter.

4.5.4.3. Defaults

Actual parameters that have default values may be indicated by an asterisk, no expression - just the next comma, or by deleting all reference to such parameters that appear at the end of the ordered parameter list.

4.5.4.4. Call by Value Restrictions

Procedure identifiers may not be called by value.

4.6. Process Statements

4.6.1. Syntax

```

<process statement> ::= SPAWN (<procedure statement>)
    | APPEND (<procedure statement>)
    | INITIATE (<procedure statement>,<process>)
    | TERMINATE
    | TERMINATE (<process>)
    | SUSPEND (<process>)
    | RESTART (<process>)

```

4.6.2. Examples

```

APPEND (PQ(1))
INITIATE (USER_JOB, P[N])
SUSPEND (P[k])
RESTART (PROCESS_23)
TERMINATE

```

4.6.3. Semantics

Process statements allow one to execute and monitor several asynchronous processes.

In order to understand the process statements, one must understand the OSL/2 philosophy. The current program in execution is an operating system that exists on top of one or more other operating systems and is capable of initiating new programs for which it may be the operating system.

The SPAWN command tells the operating system which supports the current program to run a new and separate program in a manner identical to the current program. SPAWN just enters a new job into the system. Any parameters passed to a spawned procedure are called by value.

APPEND initiates a procedure that is part of the current program. This procedure can address any variables global to it simultaneously with the current program. Control cannot leave the block that appended the procedure until all such appended procedures in that block have returned.

INITIATE initiates a new program but allows the current program to create new primitives and handle the existing operating system functions for the new program. Each such program is identified by a process variable. If the initiating program chooses not to redefine storage allocation, or any other primitive for the initiated process, then those functions are by default handled by the operating systems which support the initiating program.

SUSPEND and RESTART perform the obvious functions for their specific processes.

TERMINATE terminates the current program if no process is specified. Otherwise, the specified process will be terminated.

4.7. Iterative Statements

4.7.1. Syntax

<iterative statement> ::= WHILE <Boolean expression> DØ <statement>

| DØ <statement> UNTIL <Boolean expression>

| FØR <integer expression> TIMES DØ <statement>

| FØR <integer variable> ← <integer for list> DØ <statement>

| FØR <variable> ← <expression list> DØ <statement>

| <label> : <iterative statement> : <label>

<integer for list> ::= <for list element> | <integer for list>, <for list element>

<for list element> ::= <integer expression>

| <integer expression> TØ <integer value>

| <integer expression> TØ <integer value> BY <integer value>

| <integer expression> STEP <integer value> WHILE <Boolean expression>

| <integer expression> STEP <integer value> UNTIL <Boolean expression>

| <integer value> WHILE <Boolean expression>

| <integer value> UNTIL <Boolean expression>

<integer value> ::= <integer expression> | VALUE (<integer expression>)

4.7.2. Examples

```
WHILE P = B DØ
```

```
BEGIN
```

```
    I ← I + 1;
```

```
    B ← PIT (A[I]);
```

```
END
```

```
FØR A ← "CAT", "DØG", "PEØPLE" DØ MATCH (A)
```

```
FØR B ← 0 TO N BY 1 DØ A[B] ← "NULL"
```

```
DØ A ← A + 1 UNTIL S.[A:1] = "A"
```

4.7.3. Semantics

Iterative statements will cause a statement to be executed zero, one, or more times. Let S be a statement, B, a Boolean expression, I, an integer expression, V, a variable, and E, any expression. Then the form and semantic action associated with each iterative statement is as follows:

form 1: WHILE B DØ S

- 1) evaluate B.
- 2) if B is false, skip S and continue execution at the next sequential statement.
- 3) if B is true, execute S and repeat this sequence starting at 1).

form 2: DØ S UNTIL B

- 1) execute S.
- 2) evaluate B.
- 3) if B is false, repeat this sequence starting at 1).
- 4) if B is true, continue execution at the next sequential statement.

form 3: FØR I TIMES DØ S

- 1) evaluate I.
- 2) execute S I times.

form 4: FØR V ←

- a) I,
- b) I_1 TØ I_2 ,
- c) I_1 TØ I_2 BY I_3 ,
- d) I_1 STEP I_2 WHILE B,
- e) I_1 STEP I_2 UNTIL B,

f) I WHILE B,

g) I UNTIL B

DØ S

4a: 1) $V \leftarrow I$.

2) execute S.

3) continue execution with the next for list element.

4b: 1) $V \leftarrow I_1$.

2) if $V > I_2$ skip S and continue execution with the next for list.

3) execute S.

4) $V \leftarrow V + 1$.

5) repeat sequence 4b starting at 2).

4c: 1) $V \leftarrow I_1$.

2) if $I_3 \geq 0$.

2.1) if $V > I_2$ then start executing with the next for list element.

2.2) if $V \leq I_2$ then execute S.

2.3) skip to 4).

3) if $I_3 < 0$.

3.1) if $V < I_2$ then start executing with the next for list element.

3.2) if $V \geq I_2$ then execute S.

4) $V \leftarrow V + I_3$.

5) repeat sequence 4c starting at 2).

4d: 1) $V \leftarrow I_1$.

2) if B then execute S otherwise continue execution with the next for list element.

3) $V \leftarrow V + I_2$.

4) repeat sequence 4d starting at 2).

4e: as in 4d with B replaced by $\neg B$.

4f: 1) $V \leftarrow I$.

2) if B then execute S otherwise skip S and continue execution with the next for list element.

3) repeat sequence 4f starting at 1.

4g: as in 4f with B replaced by B.

Once all for list elements have been exhausted control continues with the statement following S.

If an expression is qualified by value, e.g.,

$\text{FOR } I \leftarrow A + B \text{ TO VALUE } (C * D/3) \text{ BY } 10$

then the integer expression qualified by value (in this case it is $C * D/3$) is evaluated only once at the beginning of the loop. This value is stored in a temporary location and used for loop calculations.

4.8. Escape Statements

4.8.1. Syntax

`<escape statement> ::= LEAVE <scope> | LEAVE <scope> (<expression>)`

`<scope> ::= <label> | <procedure identifier> | <trap identifier>`

4.8.2. Examples

LEAVE A

LEAVE FCNT (A*2)

LEAVE INNERLOOP

4.8.3. Semantics

Escape statements allow control to leave a procedure, trap condition, or one or more nested blocks, compound statements, conditionals, or iteration statements without executing all the statements following in the particular construct being executed.

A procedure identifier following a leave causes control to leave the indicated procedure body. If the procedure is used as a function, a value may be returned.

The label following a LEAVE indicates that control is to escape the range of the statement with that label. The escape expression must be contained within the statement whose label is specified. One may use a labelled LEAVE to escape one or more iterations, blocks, compounds, or conditional statements.

If a trap identifier follows a leave then the indicate trap condition is left and control continues where it left off. A value may be returned.

4.9. Pattern Matching Statements

4.9.1. Syntax

`<pattern matching statement> ::= <string> <pattern part>`

`| <variable> <pattern part>`

`| <variable> <pattern part> ← <string expression>`

`<pattern part> ::= <pattern expression> | FIXED <pattern expression>`

4.9.2. Examples

`"ABCD" PATT`

`"ABCD" {"C"}`

`STRING FIXED PATT`

`VAR {{X ← {"A" | "B" | "CAT" | PATT}} & "L"} ← MT`

4.9.3. Semantics

A pattern statement matches a pattern against a string or string variable. If a match is made, the value of the statement is true. Otherwise the statement is false. A pattern statement may be used as a Boolean expression.

If a successful match is made, that part of the string matched by the pattern may be replaced by the value of the optional string expression in the syntax above (4.10.1).

The pattern matching algorithm proceeds from left to right and attempts to match the innermost and leftmost patterns first. If two patterns are concatenated ("&") then that defines one pattern which consists of the first pattern immediately followed by the second pattern. For example, the pattern "A" & "B" is equivalent to the pattern "AB".

If a pattern alternative list is specified the pattern alternative are tried one by one from left to right until one matches or the list is exhausted.

If at any time a match is made to a pattern secondary, the value of the matched part of the string may be immediately assigned to a string variable. For example, the pattern $\{S \leftarrow \{ "A" \mid "B" \mid "12" \}\}$, if it matches at any time during the execution of the pattern matching algorithm, will necessarily assign the value "A", "B", or "12" to string variable S. Note: since subpatterns may match even if an entire pattern does not match, this may lead to serious side effects.

Normally, a pattern match is made at the left most possible position anywhere in a string. If a word FIXED preceeds the pattern in the pattern statement, then the pattern must match beginning at the first character in the string.

Within these constraints, pattern matching is recursively defined and proceeds generally left to right and from inner-most to outer-most nested patterns.

4.10. Synchronization Statements

4.10.1. Syntax

```
<synchronization statement> ::= P(<semaphore>)
                               | V(<semaphore>)
<semaphore> ::= <identifier> | <identifier> <subscript>
```

4.10.2 Examples

P(sem)

V(sem)

4.10.3.

Semaphores are used to synchronize processes. All semaphores are initialized at declaration time - normally to zero or one. When a process performs a P operation, the indicated semaphore is decremented by one. If the value of the semaphore is greater than or equal to zero, then the process continues. If the semaphore is negative, then the process is blocked and is booked on a waiting list for that semaphore.

A V operation adds one to a semaphore. If the semaphore is greater than zero no other action is taken. If the semaphore is less than or equal to zero then one process is removed from the waiting list for that semaphore. The removed process is no longer blocked and will continue execution at some time in the future.

4.11. Conditional Statements

4.11.1. Syntax

```

<conditional statement> ::= IF <Boolean expression> THEN <statement list> FI
    | IF <Boolean expression> THEN <statement list>
      ELSE <statement list> FI
    | <label> : <conditional statement> : <label>

```

4.11.2. Examples

```

IF A < B THEN R ← R + 1 FI

ØUTER: IF P THEN

BEGIN INNER: WHILE TRUE DØ

    BEGIN

        X ← X + 1;

        IF A ← X >= 3 THEN LEAVE ØUTER

        ELSE A ← X ← R * (X ← X + 1) FI;

        IF A ← X - 1 >= 3 THEN LEAVE INNER FI;

        X ← X + 1

    END: INNER

ELSE X ← 0 FI: ØUTER

```

4.11.3. Semantics

Conditional expressions may cause some statements to be skipped depending upon the value of the Boolean expression in the if clause.

If the Boolean expression is true, the statement list immediately following the THEN is executed. Control then skips over all remaining portions of the conditional statement, if any, and resumes immediately following the FI.

If the Boolean expression is false, the statement list immediately following the THEN is skipped and control continues at the next sequential

statement after the FI if no ELSE is present, or at the statement following the ELSE if an ELSE is present.

5. Declarations

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes.

Dynamically this implies the following: at the time of an entry into a block (through the BEGIN) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers have already been defined by other declarations outside, they are given a new significance for the time being. Identifiers which are not declared for the block retain their old meaning.

At the time of exit from a block (through the END, or by an escape statement) all identifiers which are declared for the block lose their local significance.

A declaration may be marked with an additional declarator:

$\emptyset$ WN or $\emptyset$ NDE. $\emptyset$ WN and $\emptyset$ NDE have the following effect:

$\emptyset$ WN: The dimensions, sizes, and string lengths of quantities declared to be own must be fixed value expressions - that is, they must be calculable at compile time. Upon re-entry into a block, the values of own quantities will be unchanged from their last exit. In contrast, the values of declared variables that are not own, and are not initialized in their declaration, are undefined.

$\emptyset$ NDE: $\emptyset$ NDE quantities are not available for use until allocated within a block. They may be allocated or released an indefinite number of times in any block. Whenever a node is released, the default node (that is, one not specifically pointed to) is the most recently

allocated node. Therefore, nodes may be used as stacks. When used as stacks, push corresponds to allocate and pop to release.

Apart from the standard functions (cf. 3.2.4., Standard Functions) and labels, all identifiers of a program must be declared. No identifier may be declared more than once in any block head.

Syntax

```

<declaration> ::= <integer declaration> | <Boolean declaration>
                | <string declaration> | <semaphore declaration>
                | <process declaration> | <pointer declaration>
                | <structure declaration> | <trap declaration>
                | <pattern declaration> | <access definition>
                | <access declaration> | <procedure declaration>
                | <file declaration> | <define declaration>

```

5.1. Integer Declarations

5.1.1. Syntax

`<id part> ::= <identifier> | (<id list>)`

`<id list> ::= <identifier> | <id list>, <identifier>`

`<number of bits> ::= <integer expression>`

`<array part> ::= <empty> | [<bound pair list>]`

`<bound pair list> ::= <bound pair> | <bound pair list>, <bound pair>`

`<bound pair> ::= <lower bound>:<upper bound>`

`<lower bound> ::= <integer expression>`

`<upper bound> ::= <integer expression>`

`<initialization> ::= <id part> <array part>`

`| <id part> <array part> ← <expression>`

`<initialization list> ::= <initialization>`

`| <initialization list>, <initialization>`

`<storage class> ::= <empty> | ØWN | NØDE`

`<integer declaration> ::= <storage class> INTEGER <initialization list>`

5.1.2. Examples

INTEGER A,B,C, (X,Y,Z) [0:10] ← 0

INTEGER A ← 0, B ← 1, (C,D) ← 3

NØDE INTEGER X ← 3, Y(32), Z(16)

5.1.3. Semantics

Identifiers declared to be integer can only take on integer values.

If a bound pair list is specified, then the corresponding identifiers all have the specified number of array indices with the specified bounds.

Initialization is performed at allocation time if specified. Thus, each time node X is allocated in the above example, it is automatically set to the value 3.

5.2. Boolean Declarations

5.2.1. Syntax

<Boolean declaration> ::= <storage class> BOOLEAN <initialization list>

5.2.2. Examples

```
BOOLEAN A,B,C
```

```
OWN BOOLEAN HAVE_I_COME_HERE_BEFORE ← FALSE
```

5.2.3. Semantics

Boolean identifiers can only take on Boolean values. Other than that their declaration is analogous to integers.

5.3. String Declarations

5.3.1. Syntax

```

<string parameter part> ::= <empty> | (<length>) | (<length>,<size>)
<length> ::= * | * <integer expression> | <integer expression>
<size> ::= <integer expression>
<string specification> ::= <id part> <string parameter part> <array part>
<string initialization> ::= <string specification>
    | <string specification> ← <string expression>
<string initialization list> ::= <string initialization>
    | <string initialization list>,<string initialization>
<string declaration> ::= <storage class> STRING <string initialization list>

```

5.3.2. Examples

```
STRING A,B,C"
```

```
STRING X(20,8),(Y,Z)(*10,3) ← MT
```

5.3.3. Semantics

String variables take on string values. Their declaration is analogous to integers. The default string parameters are maximum length = 32 characters, size = 8 bits, and variable length. I.e., if size is not specified, 8 is assumed, and if length is not specified, *32 is assumed. An asterisk ("*") prior to the length indicates that the string is variable length within the constraints of the maximum length specification.

5.4. Semaphore Declarations

5.4.1. Syntax

<semaphore declaration> ::= SEMAPHØRE <initialization list>

5.4.2. Examples

SEMAPHØRE E1, PRINTER-FINISHED ← 1,

SEMAPHØRE E[20:25], P23

5.4.3. Semantics

Semaphore declarations define semaphores for use in synchronization statements. Unless otherwise initialized, all semaphores are set to zero at declaration time.

5.5. Process Declarations

5.5.1. Syntax

```
<process declaration> ::= PRØCESS <id part> <array part>  
      | <process declaration>,<id part> <array part>
```

5.5.2. Examples

```
PRØCESS  A,B
```

```
PRØCESS  P[0:15]
```

5.5.3. Semantics

Processes are declared analogous to integers. Process identifiers are used to label processes for process statements.

5.6. Pointer Declarations

5.6.1. Syntax

<pointer declaration> ::= <storage class> PØINTER <initialization list>

5.6.2. Examples

```
PØINTER P1,P2[0:3]
```

```
PØINTER PTR30 ← NULL
```

5.6.3. Semantics

Pointers take on pointer values and are declared in a manner analogous to integers. Pointers may not be own.

5.7. Structure Declarations

5.7.1. Syntax

```

<declaration list> ::= <declaration> | <declaration list>;<declaration>

<structure part> ::= (<declaration list>)

<structure element> ::= <id part> <structure part> <array part>

<structure element list> ::= <structure element>
    | <structure element list>,<structure element>

<structure declaration> ::= <storage class> STRUCTURE <structure element list>

```

5.7.2. Examples

```

STRUCTURE PAYROLL (STRING NAME; INTEGER SALARY)

NØDE STRUCTURE LISTELEMENT (STRING ENTRY;
    STRUCTURE PTRS (PØINTER BACK, LEFT, RIGHT))

```

5.7.3. Semantics

Structure identifiers allow whole data structures to be referred to by one identifier. Each individual part of the structure may be used in the manner that its data type indicates.

5.8. Trap Declarations

5.8.1. Syntax

```
<trap declaration> ::= ØN <trap identifier> <formal parameter part>
                        DØ <statement>
```

5.8.2. Examples

```
ØN INDEX DØ TERMINATE
```

```
ØN DIVIDE-BY-ZERØ (NUMERATØR) DØ
```

```
    LEAVE DIVIDE-BY-ZERØ (CASE NUMERATØR ØF 0,1,99999)
```

5.8.3. Semantics

Trap declarations specify what action is to be taken when a trap occurs. When the indicated trap is caused the statement following the "DØ" is executed. Upon leaving the trap, control is returned where the trap occurred. Trap conditions and the parameters passed are installation dependent. There are four traps that OSL/2 can inherently generate: index out of range, divide by zero, integer overflow, and convert failure.

5.9. Pattern Declarations

5.9.1. Syntax

<pattern declaration> ::= <storage class> PATTERN <initialization list>

5.9.2. Examples

```
PATTERN ALPHABETIC ← SPAN ("ABC")
```

```
PATTERN X,Y,Z[1:3]
```

5.9.3. Semantics

Patterns are declared in a manner analogous to integers. Identifiers declared to be patterns may only take on patterns as their values.

5.10. Access Definitions

5.10.1. Syntax

<access definition> ::= <access> <access type>

<access formal parameter part>;<access definition body>

<access> ::= ACCESS | QUEUE | STACK | TABLE

<access type> ::= <identifier>

<access formal parameter part> ::= <empty>

| <access formal parameters>;<access formal parameter specification>

<access formal parameters> ::= (<formal parameter list>)

| (<formal parameter list>)[<array parameter list>]

| [<array parameter list>]

<formal parameter list> ::= <id list>

<array parameter list> ::= <array parameter>

| <array parameter list>,<array parameter>

<array parameter> ::= <identifier>

| <array parameter>:<identifier>

<access formal parameter specification> ::= <access formal parameter declaration>

| <access formal parameter specification>,<access formal parameter declaration>

<access formal parameter declaration> ::= <integer declaration>

| <Boolean declaration> | <string declaration>

| <pointer declaration> | <structure declaration>

| <pattern declaration>

<access definition body> ::= BEGIN <prefix declaration list>;

<procedure declaration list> <access end>

```

<access end> ::= END | ; END

<prefix declaration list> ::= <prefix declaration>
    | <prefix declaration list>;<prefix declaration>

<prefix declaration> ::= <integer declaration>
    | <Boolean declaration> | <string declaration>
    | <pointer declaration> | <structure declaration>
    | <pattern declaration> | <semaphore declaration>

<procedure declaration list> ::= <procedure declaration>
    | <access procedure declaration>
    | <procedure declaration list>;<procedure declaration>
    | <procedure declaration list>;<access procedure declaration>

<access procedure declaration> ::= <procedure type> <access type>
    PROCEDURE <identifier> <formal parameter part>
    <array parameter part>;<name or value>
    <formal declaration list> <procedure body>

<array parameter part> ::= <empty> | [<id list>]

```

5.10.2. Examples

```

QUEUE FIFØ (DEFAULT); INTEGER DEFAULT ← 0;

BEGIN

    INTEGER DV ← DEFAULT, QLENGTH ← 0;

    PØINTER (HEAD, TAIL) ← NULL;

    NØDE STRUCTURE ENTRY (INTEGER DATA;

                                PØINTER NEXT);

    FIFØ PRØCEDURE STØRE(X); INTEGER X;

    BEGIN

        TAIL ← TAIL.ENTRY.NEXT ← ALLØCATE (ENTRY);

        QLENGTH ← *+1;

        ENTRY.DATA ← X;

    END STØRE;

    INTEGER FIFØ PRØCEDURE FETCH [N]; INTEGER N ← 0;

    IF N <= 0

    THEN

        IF QLENGTH=0 THEN FETCH ← DV

        ELSE

            BEGIN

                PØINTER H ← HEAD;

                FETCH ← HEAD.ENTRY.DATA;

                HEAD ← HEAD.ENTRY.NEXT;

                QLENGTH ← *-1;

                RELEASE (H.ENTRY)

            END FI

```

```

ELSE
    IF QLENGTH < N THEN FETCH ← DV
    ELSE
        BEGIN
            PØINTER T ← HEAD;
            FØR N TIMES DØ
                BEGIN
                    FETCH ← T.ENTRY.DATA;
                    T ← T.ENTRY.NEXT
                END
            END
        END
    FI;
    INTEGER FIFØ PROCEDURE LENGTH; LENGTH ← QLENGTH;
END FIFØ
QUEUE MESSAGE (MSIZE) [N]; INTEGER MSIZE ← 1, N ← 20;
BEGIN
    STRING MESSAGES (MSIZE,1)[0:N-1];
    SEMAPHØRE INPUT ← N, ØOUTPUT ← 0;
    INTEGER (IN,ØUT) ← 0, MV ← N;
    STRING (MSIZE,1) MESSAGE PRØCEDURE FETCH;
    BEGIN
        P (ØOUTPUT);
        FETCH ← MESSAGES [ØUT ← (*+1) MØD MV];
        V (INPUT)
    END FETCH;
    MESSAGE PRØCEDURE STØRE(X); STRING X (MSIZE,1);

```

BEGIN

P (INPUT);

MESSAGES [IN $\leftarrow$ (*+1) MØD MV] $\leftarrow$ X;

V (ØUTPUT)

END STØRE

END MESSAGE

5.10.3. Semantics

Access definitions define new access types for later use in access declarations. Access variables are declared by an access declaration. Access definitions define the format of access declarations and define the semantics associated with an appearance of an access variable in an OSL/2 program.

The access definition declares those formal parameters which may be used in a corresponding access declaration. Parameters may be passed for use at declaration time via a normal parameter list enclosed in parentheses or via a parameter list enclosed in square brackets (i.e. array parameter list) or both. Normal procedure parameter defaults and conventions are allowed with the exception that key word parameters may not be used in the array part enclosed in square brackets.

Each new access type must have a corresponding access declaration. All variables declared in the prefix declaration list may only be referenced by procedures in the procedure declaration list. For each instance of an access variable, a complete set of variables in the prefix declaration list is allocated. This is equivalent to the insertion of the prefix declaration list in place of the access variable declaration with suitable systematic changes of the identifiers to insure that the prefix declaration list identifiers do not conflict with existing identifiers and that they may only be manipulated by the procedures in the procedure declaration list.

Procedure declarations appearing within the procedure declaration list are defined to only be within the scope of the other procedures in the procedure declaration list. Access procedure declarations declare

procedures which may be referenced external to the procedure declaration list and within the scope of the corresponding access variable. Access procedures are indicated in the access definition by repeating the identifier of the access type being defined prior to the use of the reserved word "PROCEDURE" in the procedure declaration.

When access procedures are invoked, an additional parameter, not present in the formal parameter list, is added to the actual parameter list. This parameter appears first in the actual parameter list and must be an access variable of the same type as the access procedure. Each access variable is thus matched only to the access procedure of the same type. If the access variable is subscripted, the subscript list is passed to the procedure through the array parameter part of the access procedure declaration. The parameters passed through the subscript list obey all rules of procedure actual parameters and may be passed by key word.

Two access procedures for each access type have special semantic interpretations. The procedure named "fetch" is invoked automatically when an access variable is encountered in an expression evaluation such as the right hand side of an assignment statement. The type of procedure "fetch" defined the type of the access variable on the right hand side of an assignment statement. If "fetch" has any parameters, they must all be defaulted. The procedure "store" is not typed and is invoked when the access variable appears on the left hand side of an assignment statement. The first parameter in "store" defines the type of the value stored into an access variable. All other parameters must be defaulted. Both "fetch" and "store" may be explicitly invoked as a normal function or procedure call in the OSL/2 code.

5.11. Access Declarations

5.11.1. Syntax

<access declaration> ::= <access type> <access> <access specification list>

<access specification list> ::= <access specification>

 | <access specification list>,<access specification>

<access specification> ::= <id part> <access standard parameter part>

 <access array parameter part>

<access standard parameter part> ::= <empty> | (<actual parameter list>)

<access array parameter part> ::= <empty>

 | [<access array parameter list>]

<access array parameter list> ::= <access array parameter>

 | <access array parameter list>,<access array parameter>

<access array parameter> ::= <parameter>

 | <access array parameter>:<parameter>

5.11.2. Examples

FIFØ QUEUE A,B,C

CØMPLEX TABLE (X,Y)(NUMBER_ØF_PØINTERS:22)

SPECIAL ACCESS X(15,2⁴)[0:2:100,SQRT]

5.11.3. Semantics

For each identifier in the access declaration id part, an instance of an access variable is declared. Parameters are processed as described in Section 5.10.3.

5.12. Procedure Declarations

5.12.1. Syntax

<formal parameter list> ::= <id list>

<name or value> ::= <value part> <name part> | <name part> <value part>

<value part> ::= VALUE <id list>; | <empty>

<name part> ::= NAME <id list>; | <empty>

<formal parameter part> ::= (<formal parameter list>) | <empty>

<formal parameter declaration> ::= <integer declaration>

| <Boolean declaration> | <string declaration>

| <semaphore declaration> | <process declaration>

| <pointer declaration> | <structure declaration>

| <pattern declaration> | <access declaration>

| <procedure declaration>

<formal declaration list> ::= <empty> | <formal parameter declaration>;

| <formal declaration list> <formal parameter declaration>;

<primitive or procedure> ::= PRIMITIVE | PRØCEDURE

<procedure type> ::= INTEGER | BØØLEAN | STRING <string parameter part>

| PØINTER | STRUCTURE <structure part> | PATTERN | <empty>

<procedure heading> ::= <procedure type> <primitive or procedure> <identifier>

<formal parameter part>; <name or value> <formal declaration list>

<procedure body> ::= <statement> | FØRWARD | FØRMAL

<procedure declaration> ::= <procedure heading> <procedure body>

5.12.2. Examples

PRØCEDURE ADDØNE; X ← X + 1

STRING PRØCEDURE A(S23); STRING S23;

```
BEGIN
```

```
    STRING S1, S2  $\leftarrow$  MT;
```

```
    WHILE S23 FIXED {S1  $\leftarrow$  {"A" | "B" | "C"}}  $\leftarrow$  MT DØ
```

```
    S2  $\leftarrow$  S1 & S2;
```

```
END
```

```
INTEGER PROCEDURE ADD(X); INTEGER X  $\leftarrow$  1; A  $\leftarrow$  A + X
```

5.12.3. Semantics

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which the procedure declaration appears. Associated with the body are identifiers to represent formal parameters. Formal parameters in the procedure body will, whenever the procedure is activated (cf. 3.2. Function Designators and 4.5. Procedure Statements) be assigned the values of or replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or non local to the body depending on whether they are declared within the body or not. Those which are non local to the body may well be local to the block in the head of which the procedure declaration appears. The procedure body always acts like a block, whether it has the form of one or not. If the identifier of a formal parameter is declared anew within the procedure body, it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

When a procedure declaration appears in a procedure heading to declare a formal parameter, the procedure body is FORMAL. To allow for one

pass compilation, a procedure body may be FORWARD. Thus the compiler is notified of the formats of the procedure parameters but the actual declaration of the procedure with a statement procedure body is deferred.

Formal parameters may receive default values. If the formal parameter declaration is initialized, the initializing expression is taken as a default.

5.12.4. Values of Function Designators

For a procedure declaration to define the value of a function designator there must, within the procedure body, occur one or more assignments to the function identifier or an expression must be explicitly specified in an escape statement. If no such expression is supplied, the last assignment to the function identifier is used.

Primitive procedures are within the scope of initiated processes.

5.12.5. Specifications

In the heading, a specification part, giving information about the kinds and types of the formal parameters must be included. In this part no formal parameter may occur more than once. Specifications of formal parameter must be supplied. Types, array dimensions, bases, and lengths of all formal parameters are required in the specification part.

5.13. File Declarations

5.13.1. Syntax

```

<file declaration> ::= FILE <identifier> <input output part>

                        <access technique> <record format> <system parameters>

<input output part> ::= INPUT | ØOUTPUT | <empty>

<access technique> ::= SEQUENTIAL | INDEXED |

                        <empty>

<record format> ::= (<declaration>)

                        (<declaration>,<blocking>)

                        (<declaration>,<blocking>,<number of records>)

<blocking> ::= <integer expression>

<number of records> ::= <integer expression>

<system parameters> ::= [<actual parameter list>] | <empty>

```

5.13.2. Examples

```

FILE CARDS INPUT SEQUENTIAL (STRING A(80),20)

FILE TTY SEQUENTIAL (STRING A(80)) ["NBUFR=2,UNIT=TTY32"]

```

5.13.3. Semantics

Files reside on secondary storage. Their declaration is normally very device dependent. FILE declarations in OSL/2 provide the compiler with the basic information desirable to check the source code. Details of the file which are dependent upon a particular system are in the system parameters.

Reading or writing from or to a file is done with simple assignment statements to and from simple variables, arrays, and structures of the same type and format of the record format declaration.

6. Compile Time Facilities

The dollar sign, "\$", indicates a compile time facility. It is possible to declare compile time variables and execute most OSL/2 statements at compile time.

6.1. Compile Time Declarations

6.1.1. Syntax

```

<define declaration> ::= DEFINE <define specification list>

<define specification list> ::= <define specification>
    | <define specification list>,<define specification>

<define specification> ::= <identifier> <define parameter list>
    = <define body> #

<define parameter list> ::= <id list>

<define body> ::= {any text including balanced DEFINE...# brackets
    but not including a lone#}

<compile time declaration> ::= $ <declaration>

<compile time statement> ::= $ <statement>

```

6.1.2. Examples

```

DEFINE  add(A,B) = ($A) + ($B)#

DEFINE  XX = X ← X + 2#

$INTEGER  X,Y,Z ← 0, Q ← 3

$IF  $X = $Y $THEN P ← 0 $ELSE P ← 3 $FI

```

6.1.3. OSL/2 defines are identical to B5500 defines [23].

The define declaration associates a text string with the define identifier. The later appearance of the define identifier in the OSL/2 code will cause the defined text to replace the identifier and that text will then be compiled. If actual parameters are appended to the define, then each occurrence of the corresponding formal parameter in the text will be replaced by the actual parameter. Formal parameters in the text of the define body must be preceded by "\$". The actual parameters passed to

a parameterized define follow the identifier and may be enclosed in either parentheses or square brackets. Each actual parameter is separated by a comma.

If a variable declaration is preceded by a "\$", then all variables so declared are available for use at compile time. Their appearance later in the OSL/2 code is replaced by a constant equal to their current value. Any assignment statements with a compile time variable on the left hand side must be executable at compile time. Thus they may only consist of constants and compile time variables.

Any OSL/2 statement preceded by a "\$" is executed at compile time.

Should there be any ambiguity of meaning resulting from a compile time facility, that ambiguity must be resolved by preceeding the appropriate compile time variables or reserved words within a statement by a "\$". All ambiguous syntax will be resolved as if all non marked ambiguous variables and reserved words were not compile time variables and reserved words.

In all cases a "\$" may preceed any compile time variable or reserved word.

Appendix B

An OSL/2 Compiler

In order to provide hard knowledge of the implementation difficulties of OSL/2, a compiler was generated. The exercise of building the compiler did indeed have an impact on the language. In many places, the syntax was restructured to allow for an easier parse and to make code easier to generate.

The object code was generated for the OSL/2 machine - a stack machine with arbitrary length words. There are a variety of methods available for mapping variable length data into some fixed word size. Since this is a straight forward operation that involves considerable coding time and would add little or nothing to the language, the OSL/2 machine had an undefined word length sufficient to hold any single data item.

Whenever a complex but straight forward operation was required, it was postulated to exist as a single machine instruction. If one scans through the code generators he will notice that the operators are produced in the code listing as phrases - e.g. "DUPLICATE-TOP-OF-STACK-AND-LOAD-INTEGER" which takes the address of an integer on the top of the stack, duplicates the address and then performs a "LOAD-INTEGER" operation using the address on the top of the stack. No mnemonic codes were used in the hope that it would make the code more descriptive.

The addressing scheme used was the same one described in Chapter 4. Level and displacement couples are indicated on the code, and instructions are addressed by segment and instruction number. Each block is one

segment. Instruction addresses appear in the object listing. A simple program is included which shows the object listing format.

The compiler was built using a B5500 translator writing system which produces a bottom up Floyd Production language parser [25]. The syntax was written in TWINKLE [26] and the semantics in ISL [27]. This system, unlike recursive descent translator systems, will not accept an ambiguous language. This feature and the bottom up parsing technique helped remove ambiguities from the language and made the production of object code straight forward.

The complete OSL/2 language was not implemented. The implementation proceeded until most of the difficult points were ironed out. The final language does differ slightly from the implemented language in places where the syntax was slightly changed to make OSL/2 code more readable.

The semantics language ISL is an extension of B5500 ALGOL. While coding the system, the author attempted to provide macros that duplicated OSL/2 facilities. Unfortunately, the B5500 ALGOL macro facilities were inadequate for this task. However, the exercise of producing the code was carried out as if OSL/2 were available. In over 3000 lines of semantics there were no instances where OSL/2 constructs, such as the lack of a GO TO, were a hinderance. In fact, the concentration on OSL/2 structure produced a more clean and well defined structure to the entire code than is usually written by this programmer. Furthermore, there were many many instances where OSL/2 constructs, in particular general access variables, would have considerably helped the coding effort, reduced coding time, and improved the readability of the code.

The primary lessons learned from the implementation effort were that OSL/2 is implementable, general access variables are actually rather

straight forward to implement, the program structure imposed by OSL/2 is clean yet powerful, and the constructs of OSL/2 are useable and desirable.

OSL2 /SYNTAX LISTED BY THESIS ON JANUARY 3, 1971 AT 1:19 A.M.

```

$FSWD
LANGUAGE:OSL2)
SPECIAL SYMBOLS;COMMENT;
AN <OSL2-PROGRAM> ::= <BLOCK> OR <COMPOUND-STATEMENT>
    OR <PROCEDURE-DECLARATION> ;
<BEGIN> ::= #BEGIN / <BEGIN> <COMMENT> ;
<END> ::= #END / <END> [<COMMENT> / <+I> / <+P> ] ;
<SEMI> ::= #; / <SEMI> <COMMENT> ;
<STATEMENT-LIST> ::= LIST <STATEMENT> SEPARATOR <SEMI> ;
<COMPOUND-STATEMENT> ::= <BEGIN> LIST OF <STATEMENT>S
    SEPARATED BY <SEMI> <END>
    / <OPEN-LABEL> <COMPOUND-STATEMENT> <CLOSE-LABEL> ;
<STATEMENT> ::= <ASSIGNMENT-STATEMENT>
    / <CASE-STATEMENT>
    / <PROCEDURE-STATEMENT>
    / <ITERATIVE-STATEMENT>
    / <ESCAPE-STATEMENT>
    / <PROCESS-STATEMENT>
    / <EVENT-STATEMENT>
    / <PATTERN-MATCHING-STATEMENT>
    / <CONDITIONAL-STATEMENT>
    / <COMPOUND-STATEMENT>
    / <BLOCK>
    / <ALLOCATION-STATEMENT>
    / <DUMMY-STATEMENT> ;
<DUMMY-STATEMENT> ::= EMPTY ;
<ASSIGNMENT-STATEMENT> ::= <INTEGER-ASSIGNMENT>
    / <BOOLEAN-ASSIGNMENT>
    / <STRING-ASSIGNMENT>
    / <POINTER-ASSIGNMENT>
    / <STRUCTURE-ASSIGNMENT>
    / <PATTERN-ASSIGNMENT> ;
<ARROW> ::= #+ / #:= ;
<DECLARATION> ::= <INTEGER-DECLARATION>
    / <BOOLEAN-DECLARATION>
    / <STRING-DECLARATION>
    / <POINTER-DECLARATION>
    / <EVENT-DECLARATION>
    / <PROCESS-DECLARATION>
    / <PROCEDURE-DECLARATION>
    / <PATTERN-DECLARATION>
    / <STRUCTURE-DECLARATION>
    / <ACCESS-DECLARATION>
    / <CODE-DECLARATION>
    / <FILE-DECLARATION>
    / <ACCESS-DEFINITION>
    / <EVENT-ACTION-DECLARATION>
    / <MACRO-DEFINITION>
    / <DEFAULT-DEFINITION>;
<ID-PART> ::= <+I> @S ENTERIDINDEFCLSTACK
    /#( LIST OF [<+I> @S ENTERIDINDEFCLSTACK ] SEPARATED BY #, ) ;
<STORAGE-CLASS> ::= EMPTY / #OWN @S OWNDECLARATION
    / #NODE @S NODEDECLARATION ;
<NO-CLASS-INTEGER-DECLARATION> ::= #INTEGER @S INTEGERDECLARATION
    FOLLOWED BY LIST OF
    <INTEGER-INITIALIZATION>S SEPARATED BY COMMAS
    @S ENDOFDATADECLARATION ;

```

```

<INTEGER-DECLARATION> ::= <STORAGE-CLASS><NO-CLASS-INTEGER-DECLARATION>; 00005800
<NO-CLASS-BOOLEAN-DECLARATION> ::= #BOOLEAN @S BOOLEANDECLARATION 00005900
    FOLLOWED BY LIST OF <BOOLEAN-INITIALIZATION>S SEPARATED BY 00006000
    COMMAS @S ENDOFDATADECLARATION ; 00006100
<BOOLEAN-DECLARATION> ::= <STORAGE-CLASS><NO-CLASS-BOOLEAN-DECLARATION>; 00006200
<NO-CLASS-STRING-DECLARATION> ::= #STRING @S STRINGDECLARATION 00006300
    FOLLOWED BY LIST OF <STRING-INITIALIZATION>S SEPARATED BY 00006400
    COMMAS @S ENDOFDATADECLARATION ; 00006500
<STRING-DECLARATION> ::= <STORAGE-CLASS> <NO-CLASS-STRING-DECLARATION>; 00006600
<EVENT-DECLARATION> ::= #EVENT @S EVENTDECLARATION FOLLOWED BY LIST OF 00006700
    [<ID-PART> FOLLOWED BY <ARRAY-PART> 00006800
    @S RESETIDLISTCOUNTER] SEPARATED BY 00006900
    COMMAS @S ENDOFDATADECLARATION ; 00007000
<NO-CLASS-CODE-DECLARATION> ::= #CODE @S CODEDECLARATION FOLLOWED 00007100
    BY LIST OF [<ID-PART> FOLLOWED BY <ARRAY-PART> 00007200
    @S RESETIDLISTCOUNTER ] 00007300
    SEPARATED BY COMMAS @S ENDOFDATADECLARATION ; 00007400
<CODE-DECLARATION> ::= <STORAGE-CLASS> <NO-CLASS-CODE-DECLARATION>; 00007500
<NO-CLASS-POINTER-DECLARATION> ::= #POINTER 00007600
    @S POINTERDECLARATION FOLLOWED BY LIST OF 00007700
    <POINTER-INITIALIZATION>S SEPARATED BY COMMAS 00007800
    @S ENDOFDATADECLARATION ; 00007900
<SUBSTRUCTURE-DECLARATION> ::= 00008000
    <NO-CLASS-INTEGER-DECLARATION> 00008100
    / <NO-CLASS-BOOLEAN-DECLARATION> 00008200
    / <NO-CLASS-STRING-DECLARATION> 00008300
    / <NO-CLASS-CODE-DECLARATION> 00008400
    / <NO-CLASS-POINTER-DECLARATION> 00008500
    / <NO-CLASS-STRUCTURE-DECLARATION> 00008600
    / <NO-CLASS-PROCESS-DECLARATION> 00008700
    / <NO-CLASS-PATTERN-DECLARATION> 00008800
    / <ACCESS-DECLARATION> 00008900
    / <PROCEDURE-DECLARATION> 00009000
    / <FILE-DECLARATION> ] 00009100
<NODE-CLASS> ::= #NODE @S NODEDECLARATION / EMPTY ; 00009200
<POINTER-DECLARATION> ::= <NODE-CLASS> <NO-CLASS-POINTER-DECLARATION>; 00009300
<PROCESS-DECLARATION> ::= <NODE-CLASS> <NO-CLASS-PROCESS-DECLARATION>; 00009400
<STRUCTURE-DECLARATION> ::= <STORAGE-CLASS> 00009500
    <NO-CLASS-STRUCTURE-DECLARATION> 00009600
    @S ENDOFENTIRESTRUCTUREDECLARATION ; 00009700
<NO-CLASS-STRUCTURE-DECLARATION> ::= #STRUCTURE @S STRUCTUREDECLARATION 00009800
    LIST OF <STRUCTURE-ELEMENT>S SEPARATED BY COMMAS 00009900
    @S ENDOFDATADECLARATION ; 00010000
<STRUCTURE-ELEMENT> ::= <ID-PART> <STRUCTURE-PART> <ARRAY-PART> 00010100
    @S RESETIDLISTCOUNTER ; 00010200
<NO-CLASS-PATTERN-DECLARATION> ::= #PATTERN @S PATTERNDECLARATION 00010300
    LIST OF <PATTERN-INITIALIZATION>S SEPARATED BY COMMAS 00010400
    @S ENDOFDATADECLARATION ; 00010500
<PATTERN-DECLARATION> ::= <STORAGE-CLASS><NO-CLASS-PATTERN-DECLARATION>; 00010600
<ACCESS-DEFINITION> ::= <ACCESS-HEADING> <ACCESS-BODY> ; 00010700
<ACCESS-HEADING> ::= [ #ACCESS / #QUEUE / #STACK ] 00010800
    <*I> @S ACCESSIDENTIFIER 00010900
    <FORMAL-PARAMETER-JUNK> @S ENDOFACCESSHEADING; 00011000
<ACCESS-BODY> ::= #BEGIN [ <BASE-STRUCTURE> #] / @S NORASESTRUCTURE ] 00011100
    [ <ENTRY-STRUCTURE> #] / @S NOENTRYSTRUCTURE ] 00011200
    <ACCESS-PROCEDURE-DECLARATION-LIST>#]; #END @SENOFACCESSBODY; 00011300
<BASE-STRUCTURE> ::= <NO-CLASS-SINGLE-STRUCTURE-DECLARATION> 00011400
    @S STRUCTUREMUSTBENAMEDBASE; 00011500
<ENTRY-STRUCTURE> ::= #NODE @S NODEDECLARATION 00011600
    <NO-CLASS-SINGLE-STRUCTURE-DECLARATION> 00011700
    @S STRUCTUREMUSTBENAMEENTRY; 00011800

```

NO-CLASS-SINGLE-STRUCTURE-DECLARATION> ::= #STRUCTURE	00011900
@S STRUCTUREDECLARATION <+I> @S ENTERIDINDECLSTACK	00012000
<STRUCTURE-PART> @S RESETIDLISTCOUNTER	00012100
@S ENDOFDATADECLARATION;	00012200
ACCESS-PROCEDURE-DECLARATION-LIST> ::=	00012300
LIST [<PROCEDURE-DECLARATION>	00012400
/ <ACCESS-PROCEDURE-DECLARATION>]	00012500
SEPARATOR <SEMI>;	00012600
ACCESS-PROCEDURE-DECLARATION> ::= <ACCESS-PROCEDURE-HEADING>	00012700
@S CODEBEGINPROCEDUREBODY <STATEMENT>	00012800
@S CODEENDPROCEDUREBODY @S CHECKNOTFORMAL	00012900
@S ENDOFACCESSPROCEDUREBODY;	00013000
ACCESS-PROCEDURE-HEADING> ::= <PROCEDURE-TYPE> <+I> @S CURRENTACCESSID	00013100
#PROCEDURE <+I> @S PROCEDUREIDENTIFIER	00013200
@S ENTERACCESSPROCEDUREID	00013300
<FORMAL-PARAMETER-JUNK> @S ENDOFPROCEDUREHEADING;	00013400
ACCESS-DECLARATION> ::= <ACCESS-TYPE> @S ACCESSDECLARATION	00013500
[#ACCESS / #QUEUE / #STACK]	00013600
LIST <ACCESS-PART> SEPARATOR COMMA @S ENDOFDATADECLARATION;	00013700
ACCESS-TYPE> ::= <+I> @T ISIDANACCESSTYPE;	00013800
PROCEDURE-HEADING> ::= <PROCEDURE-TYPE> #PROCEDURE	00013900
<+I> @S PROCEDUREIDENTIFIER	00014000
<FORMAL-PARAMETER-JUNK> @S ENDOFPROCEDUREHEADING ;	00014100
PROCEDURE-TYPE> ::= EMPTY	00014200
/ #CODE @S CODEDECLARATION	00014300
/ #INTEGER @S INTEGERDECLARATION	00014400
/ #BOOLEAN @S BOOLEANDECLARATION	00014500
/ #STRING @S STRINGDECLARATION	00014600
[#(## <INTEGER-EXPRESSION> &	00014700
[# <INTEGER-EXPRESSION>] & #)] &	00014800
/ #POINTER @S POINTERDECLARATION	00014900
/ #STRUCTURE @S STRUCTUREDECLARATION @S PUSHDUMMYID	00015000
<STRUCTURE-PART>	00015100
/ #PATTERN @S PATTERNDECLARATION ;	00015200
FORMAL-PARAMETER-JUNK> ::= <FORMAL-PARAMETER-LIST> <SEMI>	00015300
<VALUE-PART> <FORMAL-PARAMETER-SPECIFICATION> / <SEMI> ;	00015400
FORMAL-PARAMETER-LIST> ::= #(	00015500
LIST OF[<+I> @S FORMALPARAMETER] SEPARATED BY #, #)	00015600
@S SCANNINGFORMALPARAMETERDECLARATIONS ;	00015700
VALUE-PART> ::= [#VALUE LIST OF [<+I> @S VALUEIDENTIFIER]	00015800
SEPARATED BY COMMAS <SEMI>] & ;	00015900
FORMAL-PARAMETER-SPECIFICATION> ::=	00016000
LIST OF	00016100
<FORMAL-PARAMETER-DECLARATION>S SEPARATED BY <SEMI> <SEMI>	00016200
@S ENDOFFORMALS ;	00016300
FORMAL-PARAMETER-DECLARATION> ::= <INTEGER-DECLARATION>	00016400
/ <BOOLEAN-DECLARATION>	00016500
/ <STRING-DECLARATION>	00016600
/ <EVENT-DECLARATION>	00016700
/ <CODE-DECLARATION>	00016800
/ <PROCESS-DECLARATION>	00016900
/ <POINTER-DECLARATION>	00017000
/ <STRUCTURE-DECLARATION>	00017100
/ <PATTERN-DECLARATION>	00017200
/ <ACCESS-DECLARATION>	00017300
/ <PROCEDURE-DECLARATION>	00017400
/ <FILE-DECLARATION> ;	00017500
MACRO-DEFINITION> ::= #MACRO @S MACRODEFINITION	00017600
[<NO-CLASS-INTEGER-DECLARATION>	00017700
/ <NO-CLASS-BOOLEAN-DECLARATION>	00017800
/ <NO-CLASS-STRING-DECLARATION>	00017900

```

/ <NO-CLASS-PATTERN-DECLARATION>                                00018000
/ <NO-CLASS-STRUCTURE-DECLARATION>                               00018100
/ <PROCEDURE-DECLARATION> ] ;                                    00018200
<FILE-DECLARATION> ::= #FILE @S FILEDECLARATION                 00018300
LIST OF[ <+I> @S ENTERIDINDECLSTACK ] SEPARATOR #,             00018400
@S ENDOFDATADECLARATION ;                                       00018500
<OPEN-LABEL> ::= <+I> @S OPENLABEL # ;                          00018600
<CLOSE-LABEL> ::= # ; <+I> @S CLOSELABEL ;                     00018700
<STRUCTURE-ASSIGNMENT> ::= <STRUCTURE-VARIABLE> <ARROW>        00018800
<STRUCTURE-EXPRESSION> @S CHECKTHATSTRUCTURESARETHESAME        00018900
@S CDESTRUCTURESTORE ;                                          00019000
<POINTER-ID> ::= <+I> @T TESTPOINTER @S MUSTBEUNSUBSCRIPTED    00019100
/ <+I> @T TESTPOINTER <SUBSCRIPT> @S CHECKSUBSCRIPTMATCH ;     00019200
<INTEGER-ID> ::= <+I> @T TESTINTEGER @S MUSTBEUNSUBSCRIPTED     00019300
/ <+I> @T TESTINTEGER <SUBSCRIPT> @S CHECKSUBSCRIPTMATCH ;     00019400
<BOOLEAN-ID> ::= <+I> @T TESTBOOLEAN @S MUSTBEUNSUBSCRIPTED     00019500
/ <+I> @T TESTBOOLEAN <SUBSCRIPT> @S CHECKSUBSCRIPTMATCH ;     00019600
<STRING-ID> ::= <+I> @T TESTSTRING @S MUSTBEUNSUBSCRIPTED       00019700
/ <+I> @T TESTSTRING <SUBSCRIPT> @S CHECKSUBSCRIPTMATCH ;      00019800
<STRUCTURE-ID> ::= <+I> @T TESTSTRUCTURE @S MUSTBEUNSUBSCRIPTED 00019900
/ <+I> @T TESTSTRUCTURE <SUBSCRIPT> @S CHECKSUBSCRIPTMATCH ;    00020000
<PATTERN-ID> ::= <+I> @T TESTPATTERN @S MUSTBEUNSUBSCRIPTED     00020100
/ <+I> @T TESTPATTERN <SUBSCRIPT> @S CHECKSUBSCRIPTMATCH ;     00020200
<EVENT-ID> ::= <+I> @T TESTEVENT @S MUSTBEUNSUBSCRIPTED         00020300
/ <+I> @T TESTEVENT <SUBSCRIPT> @S CHECKSUBSCRIPTMATCH ;       00020400
<PROCESS-ID> ::= <+I> @T TESTPROCESS @S MUSTBEUNSUBSCRIPTED     00020500
/ <+I> @T TESTPROCESS <SUBSCRIPT> @S CHECKSUBSCRIPTMATCH ;     00020600
<CODE-ID> ::= <+I> @T TESTCODE @S MUSTBEUNSUBSCRIPTED           00020700
/ <+I> @T TESTCODE <SUBSCRIPT> @S CHECKSUBSCRIPTMATCH ;         00020800
<FILE-ID> ::= <+I> @T TESTFILE @S MUSTBEUNSUBSCRIPTED           00020900
/ <+I> @T TESTFILE <SUBSCRIPT> @S CHECKSUBSCRIPTMATCH ;        00021000
<STRUCTURE-FIRST-PART> ::= LIST [ <STRUCTURE-ID>                00021100
@S SETCURRENTSTRUCTUREPTR ] SEPARATOR #, ;                      00021200
<POINTER-NAME> ::= [ <STRUCTURE-FIRST-PART> #, ] & <POINTER-ID> @S RESETSPT; 00021300
<PATTERN-NAME> ::= [ <STRUCTURE-FIRST-PART> #, ] & <PATTERN-ID> @S RESETSPT; 00021400
<INTEGER-NAME> ::= [ <STRUCTURE-FIRST-PART> #, ] & <INTEGER-ID> @S RESETSPT; 00021500
<BOOLEAN-NAME> ::= [ <STRUCTURE-FIRST-PART> #, ] & <BOOLEAN-ID> @S RESETSPT; 00021600
<PROCESS-NAME> ::= [ <STRUCTURE-FIRST-PART> #, ] & <PROCESS-ID> @S RESETSPT; 00021700
<STRING-NAME> ::= [ <STRUCTURE-FIRST-PART> #, ] & <STRING-ID> @S RESETSPT; 00021800
<CODE-NAME> ::= [ <STRUCTURE-FIRST-PART> #, ] & <CODE-ID> @S RESETSPT; 00021900
<FILE-NAME> ::= [ <STRUCTURE-FIRST-PART> #, ] & <FILE-ID> @S RESETSPT; 00022000
<SUBSCRIPT> ::= # [ @S STARTSUBSCRIPT LIST[ <INTEGER-EXPRESSION> 00022100
@S INCRSUBSCRIPT ] SEPARATOR #, # ] ;                          00022200
<POINTER-SPECIFIER> ::= <POINTER-NAME> #,                     00022300
/ <POINTER-SPECIFIER> <POINTER-NAME> #, ;                      00022400
<EVENT-VARIABLE> ::= <EVENT-ID> ;                               00022500
<ACCESS-ID> ::= <+I> @T IDISACCESS ;                             00022600
<INTEGER-FUNCTION> ::= <FUNCTION> @T FUNCTIONISINTEGER ;       00022700
<PATTERN-FUNCTION> ::= <FUNCTION> @T FUNCTIONISPATTERN ;       00022800
<POINTER-FUNCTION> ::= <FUNCTION> @T FUNCTIONISPOINTER ;       00022900
<STRING-FUNCTION> ::= <FUNCTION> @T FUNCTIONISSTRING ;         00023000
<BOOLEAN-FUNCTION> ::= <FUNCTION> @T FUNCTIONISBOOLEAN ;       00023100
<STRUCTURE-FUNCTION> ::= <FUNCTION> @T FUNCTIONISSTRUCTURE ;   00023200
<STRUCTURE-EXPRESSION> ::= <STRUCTURE-VARIABLE>                00023300
/ <STRUCTURE-FUNCTION>                                           00023400
/ #IF <BOOLEAN-EXPRESSION> #THEN <STRUCTURE-EXPRESSION>         00023500
#ELSE <STRUCTURE-EXPRESSION>                                     00023600
@S CHECKTHATSTRUCTURESARETHESAME #FI 00023700
@S MOVEOSL2SFMFIFLDFROM1TO6 00023800
/ #CASE <INTEGER-EXPRESSION> #OF 00023900
<STRUCTURE-EXPRESSION-LIST> #ESAC 00024000

```

```

@S MOVEOSL2SEMFIELDFROMIT04 ; 00024100
STRUCTURE-EXPRESSION-LIST> ::= <STRUCTURE-EXPRESSION> 00024200
/ <STRUCTURE-EXPRESSION-LIST> #, <STRUCTURE-EXPRESSION> 00024300
@S CHECKTHATSTRUCTURESARETHESAME ; 00024400
DEFAULT-DEFINITION> ::= #DEFAULT 00024500
[ #INTEGER #( <INTEGER-EXPRESSION> #) 00024600
/ #INTEGER # = [#ONES/#TWS] 00024700
/ #INTEGER #( <INTEGER-EXPRESSION> #) # = [#ONES/#TWS] 00024800
/ #STRING #( <LENGTH> #, <SIZE> #) [# = <STRING-CODE> ] & 00024900
/ #STRING # = <STRING-CODE> ] 00025000
STRING-CODE> ::= #BCL /#BCD /#BCDIC /#ASCII7 /#ASCII8; 00025100
PROCEDURE-STATEMENT> ::= <PROCEDURE-ID> @S CHECKTHATPARMSNOTREQUIRED 00025200
/ <PROCEDURE-ID> <ACTUAL-PARAMETER-LIST> ; 00025300
FUNCTION> ::= <PROCEDURE-ID> @T CHECKTHATIDISTYPED 00025400
@S CHECKTHATPARMSNOTREQUIRED 00025500
/ <PROCEDURE-ID> @T CHECKTHATIDISTYPED 00025600
<ACTUAL-PARAMETER-LIST> ; 00025700
ACTUAL-PARAMETER-LIST> ::= <ACTUAL-PARAMETER-FIRST-PART> #) 00025800
@S ENDOFACTUALPARAMETERSCAN; 00025900
ACTUAL-PARAMETER-FIRST-PART> ::= #( @S SETUPACTUALPARAMETERSCAN 00026000
[ @T CHECKIFNEXTFFISAPROCEDURE 00026100
[ <PROCEDURE-ID> @S CODEPROCEDUREIDACTUALPARAMETER 00026200
/ ##& @S CODEDEFAULTPROCEDUREIDACTUALPARAMETER ] 00026300
/ @T CHECKIFNEXTFFISACCESS 00026400
[ <ACCESS-ID> @S CODEACCESSIDACTUALPARAMETER 00026500
/ ##& @S CODEDEFAULTACCESSIDACTUALPARAMETER ] 00026600
/ @T CHECKIFNEXTFFISINTEGER 00026700
[ <INTEGER-EXPRESSION> @S CODEINTEGERACTUALPARAMETER 00026800
/ ##& @S CODEDEFAULTINTEGERACTUALPARAMETER ] 00026900
/ @T CHECKIFNEXTFFISBOOLEAN 00027000
[ <BOOLEAN-EXPRESSION> @S CODEBOOLEANACTUALPARAMETER 00027100
/ ##& @S CODEDEFAULTBOOLEANACTUALPARAMETER ] 00027200
/ @T CHECKIFNEXTFFISSTRING 00027300
[ <STRING-EXPRESSION> @S CODESTRINGACTUALPARAMETER 00027400
/ ##& @S CODEDEFAULTSTRINGACTUALPARAMETER ] 00027500
/ @T CHECKIFNEXTFFISPOINTER 00027600
[ <POINTER-EXPRESSION> @S CODEPOINTERACTUALPARAMETER 00027700
/ ##& @S CODEDEFAULTPOINTERACTUALPARAMETER ] 00027800
/ @T CHECKIFNEXTFFISPATTERN 00027900
[ <PATTERN-EXPRESSION> @S CODEPATTERNACTUALPARAMETER 00028000
/ ##& @S CODEDEFAULTPATTERNACTUALPARAMETER ] 00028100
/ @T CHECKIFNEXTFFISSTRUCTURE 00028200
[ <STRUCTURE-EXPRESSION> @S CODESTRUCTUREACTUALPARAMETER 00028300
/ ##& @S CODEDEFAULTSTRUCTUREACTUALPARAMETER ] 00028400
/ @T CHECKIFNEXTFFISEVENT 00028500
[ <EVENT-VARIABLE> @S CODEEVENTACTUALPARAMETER 00028600
/ ##& @S CODEDEFAULTEVENTACTUALPARAMETER ] 00028700
/ @T CHECKIFNEXTFFISFILE 00028800
[ <FILE-VARIABLE> @S CODEFILEACTUALPARAMETER 00028900
/ ##& @S CODEDEFAULTFILEACTUALPARAMETER ] 00029000
/ @T CHECKIFNEXTFFISPROCESS 00029100
[ <PROCESS-VARIABLE> @S CODEPROCESSACTUALPARAMETER 00029200
/ ##& @S CODEDEFAULTPROCESSACTUALPARAMETER ] 00029300
/ @T CHECKIFNEXTFFISCODE 00029400
[ <CODE-VARIABLE> @S CODECODEACTUALPARAMETER 00029500
/ ##& @S CODEDEFAULTCODEACTUALPARAMETER ] ] 00029600
@S GETNEXTPARM 00029700
/ <ACTUAL-PARAMETER-FIRST-PART> #, 00029800
[ @T CHECKIFNEXTFFISAPROCEDURE 00029900
[ <PROCEDURE-ID> @S CODEPROCEDUREIDACTUALPARAMETER 00030000
/ ##& @S CODEDEFAULTPROCEDUREIDACTUALPARAMETER ] 00030100

```

```

/ @T CHECKIFNEXTFPISACCESS                                00030200
[ <ACCESS-ID> @S CODEACCESSIDACTUALPARAMETER              00030300
/ **& @S CODEDEFAULTACCESSIDACTUALPARAMETER ]            00030400
/ @T CHECKIFNEXTFPISINTEGER                                00030500
[ <INTEGER-EXPRESSION> @S CODEINTEGFRACTUALPARAMETER      00030600
/ **& @S CODEDEFAULTINTEGERACTUALPARAMETER ]              00030700
/ @T CHECKIFNEXTFPISBOOLEAN                                00030800
[ <BOOLEAN-EXPRESSION> @S CODEROOOFANACTUALPARAMETER      00030900
/ **& @S CODEDEFAULTROOLFANACTUALPARAMETER ]              00031000
/ @T CHECKIFNEXTFPISSTRING                                 00031100
[ <STRING-EXPRESSION> @S CODESTRINGACTUALPARAMETER        00031200
/ **& @S CODEDEFAULTSTRINGACTUALPARAMETER ]              00031300
/ @T CHECKIFNEXTFPISPOINTER                                00031400
[ <POINTER-EXPRESSION> @S CODEPOINTERACTUALPARAMETER      00031500
/ **& @S CODEDEFAULTPOINTERACTUALPARAMETER ]              00031600
/ @T CHECKIFNEXTFPISPATTERN                                00031700
[ <PATTERN-EXPRESSION> @S CODEPATTERNACTUALPARAMETER      00031800
/ **& @S CODEDEFAULTPATTERNACTUALPARAMETER ]              00031900
/ @T CHECKIFNEXTFPISSTRUCTURE                              00032000
[ <STRUCTURE-EXPRESSION> @S CODESTRUCTURFACTUALPARAMETER 00032100
/ **& @S CODEDEFAULTSTRUCTUREACTUALPARAMETER ]            00032200
/ @T CHECKIFNEXTFPISEVENT                                  00032300
[ <EVENT-VARIABLE> @S CODEEVENTACTUALPARAMETER            00032400
/ **& @S CODEDEFAULTEVENTACTUALPARAMETER ]                00032500
/ @T CHECKIFNEXTFPISFILE                                    00032600
[ <FILE-VARIABLE> @S CODEFILEACTUALPARAMETER              00032700
/ **& @S CODEDEFAULTFILEACTUALPARAMETER ]                 00032800
/ @T CHECKIFNEXTFPISPROCESS                                00032900
[ <PROCESS-VARIABLE> @S CODEPROCESSACTUALPARAMETER        00033000
/ **& @S CODEDEFAULTPROCFESSACTUALPARAMETER ]             00033100
/ @T CHECKIFNEXTFPISCODE                                    00033200
[ <CODE-VARIABLE> @S CODECODEACTUALPARAMETER              00033300
/ **& @S CODEDEFAULTCODEACTUALPARAMETER ] ]              00033400
@S GETNEXTPARM ;                                           00033500
<COMMENT> ::= #COMMENT @S SKIPTHROUGHSEMICOLON ;          00033600
<BLOCK> ::= <BEGIN> @S CODEPREDECLARATIONBLOCKHEAD        00033700
LIST [ <DECLARATION> FOLLOWED BY A <SEMI> ]                00033800
@S CODEPOSTDECLARATIONBLOCKHEAD @S ENDOFDECLIST           00033900
<STATEMENT-LIST> <END> @S CODEBLOCKEND @S ENDOFBLOCK      00034000
/ <OPEN-LABEL> <BLOCK> <CLOSE-LABEL> ;                    00034100
<LOWER-BOUND> ::= <INTEGER-EXPRESSION> @S CODEFLOWERBOUND; 00034200
<UPPER-BOUND> ::= <INTEGER-EXPRESSION> @S CODEUPPERBOUND;  00034300
<INTEGER-INITIALIZATION> ::= <ID-PART> <ARRAY-PART>       00034400
[ EMPTY / <ARROW> @S CODEBEGININTEGERINITIALIZATION      00034500
<INTEGER-EXPRESSION> @S CODEENDINTEGERINITIALIZATION ]    00034600
@S RESETIDLISTCOUNTER ;                                    00034700
<ARRAY-PART> ::= EMPTY                                     00034800
/ <ARRAY-FIRST-PART> #] @S ENTERNUMBEROFSUBSCRIPTS ;      00034900
<ARRAY-FIRST-PART> ::= #[ @S CODEBEGINLOWERBOUND <LOWER-BOUND> 00035000
# : @S CODEBEGINUPPERBOUND <UPPER-BOUND>                  00035100
@S ADDANOTHERSUBSCRIPT                                     00035200
/ <ARRAY-FIRST-PART> #, @S CODEBEGINLOWERBOUND <LOWER-BOUND> 00035300
# : @S CODEBEGINUPPERBOUND <UPPER-BOUND>                   00035400
@S ADDANOTHERSUBSCRIPT ;                                    00035500
<BOOLEAN-INITIALIZATION> ::= <ID-PART> <APRAY-PART>       00035600
[ EMPTY / <ARROW> @S CODEBEGINPOOLEANINITIALIZATION       00035700
<ROOLFAN-EXPRESSION> @S CODEENDBOOLEANINITIALIZATION ]    00035800
@S RESETIDLISTCOUNTER ;                                    00035900
<LENGTH> ::= [ ** ]& @S SETDEFAULTSTRINGLENGTH            00036000
/ ** <INTEGER-EXPRESSION> @S SETVARIABLESTRINGLENGTH      00036100
/ <INTEGER-EXPRESSION> @S SETFIXEDSTRINGLENGTH ;           00036200

```

```

<SIZE> ::= <INTEGER-EXPRESSION> @S SETSTRINGSIZE ;
<STRING-SPECIFICATION> ::= <ID-PART>
    [ @S SETDEFAULTSTRINGLENGTHANDSIZE
      / #( @S CODEBEGINSTRINGLENGTH <LENGTH>
        [ #, @S CODEBEGINSTRINGSIZE <SIZE>
          / @S SETDEFAULTSTRINGSIZE ] #) ] <ARRAY-PART> ;
<STRING-INITIALIZATION> ::= <STRING-SPECIFICATION>
    [ <ARROW> @S CODEBEGINSTRINGINITIALIZATION
      <STRING-EXPRESSION> @S CODEENDSTRINGINITIALIZATION ]&
    @S RESETIDLISTCOUNTER ;
<POINTER-INITIALIZATION> ::= <ID-PART> <ARRAY-PART>
    [ <ARROW> @S CODEBEGINPOINTERINITIALIZATION
      <POINTER-EXPRESSION> @S CODEENDPOINTERINITIALIZATION ]&
    @S RESETIDLISTCOUNTER ;
<NO-CLASS-PROCESS-DECLARATION> ::= #PROCFSS @S PROCESSDECLARATION
    LIST [ <ID-PART> FOLLOWED BY <ARRAY-PART>
      @S CODEPROCESSDECLARATION @S RESETIDLISTCOUNTER ]
    SEPARATOR #, @S ENDOFDATADECLARATION ;
<STRUCTURE-PART> ::= #( @S SCANSUBSTRUCTURE @S CODEBEGINSTRUCTUREDECL
    LIST <SUBSTRUCTURE-DECLARATION> SEPARATOR <SEMI>
    #) @S CODEENDSTRUCTUREDECL @S ENDOFSTRUCTUREDECLARATION ;
<EVENT-ACTION-DECLARATION> ::= #ON <EVENT-VARIABLE>
    @S CODEBEGINEVENTACTIONDECL #DO <STATEMENT>
    @S CODEENDEVENTACTIONDECL ;
<PATTERN-INITIALIZATION> ::= <ID-PART> <ARRAY-PART>
    [ <ARROW> @S CODEBEGINPATTERNINITIALIZATION
      <PATTERN-EXPRESSION> @S CODEENDPATTERNINITIALIZATION ]&
    @S RESETIDLISTCOUNTER ;
<ACCESS-PART> ::= <ID-PART>
    [ @S MAKESURENOPARAMETERSAREREQUIRED
      / @S SETUPACCESSPARAMETERSCAN <ACTUAL-PARAMETER-LIST>
      @S ENDOFACCESSPARAMETERSCAN ] <ARRAY-PART>
    @S CODEACCESSDECLARATION @S RESETIDLISTCOUNTER ;
<PROCEDURE-DECLARATION> ::= <PROCEDURE-HEADING>
    [ @S CODEBEGINPROCEDUREBODY <STATEMENT>
      @S CODEENDPROCEDUREBODY @S CHECKNOTFORMAL
      / #FORMAL @S CHECKFORMALALLOWED
      / #FORWARD @S CHECKFORWARDOK ] @S ENDOFPROCEDUREBODY ;
<CONDITIONAL-STATEMENT> ::= #IF <BOOLEAN-EXPRESSION>
    @S CODEBRANCHPASTTHENIFFALSE #THEN
    <STATEMENT-LIST> @S CODEENDFALSEBRANCHPASTTHEN
    [ #ELSE @S CODEBRANCHPASTELSE <STATEMENT-LIST>
      @S CODEENDBRANCHPASTELSE ]& #FI
    / <OPEN-LABEL> <CONDITIONAL-STATEMENT> <CLOSE-LABEL> ;
<CASE-STATEMENT> ::= #CASE [ <INTEGER-EXPRESSION>
    / #WAIT #( @S CODECASEWAITSTATEMENT <EVENT-LIST> #)
      @S CODEENDCASEWAIT ] #OF @S CODECASEBRANCH
    LIST [ <STATEMENT> @S CODECASESTATEMENTENTRY ]
    SEPARATOR <SEMI>
    [ #DEFAULT @S CODECASESTATEMENTDEFAULT <STATEMENT>
      @S CODEENDCASESTATEMENTDEFAULT
      / @S CODENOCASESTATEMENTDEFAULT ]
    #ESAC @S CODEENDCASE
    / <OPEN-LABEL> <CASE-STATEMENT> <CLOSE-LABEL> ;
<ITERATIVE-STATEMENT> ::=
    #WHILE @S CODEBEGINWHILESTATEMENT <BOOLEAN-EXPRESSION>
    #DO @S CODEBRANCHPASTSTATEMENTIFFALSE <STATEMENT>
    @S CODEBRANCHBACKTOWHILE @S CODEENDBRANCHIFFALSE
    / #DO @S CODEBEGINDOSTATEMENT <STATEMENT> #UNTIL
      <BOOLEAN-EXPRESSION> @S CODEBRANCHBACKTODOIFFALSE
    / #FOR [ <INTEGER-VARIABLE> @S CODEFORVARIABLE <ARROW>

```

```

00036300
00036400
00036500
00036600
00036700
00036800
00036900
00037000
00037100
00037200
00037300
00037400
00037500
00037600
00037700
00037800
00037900
00038000
00038100
00038200
00038300
00038400
00038500
00038600
00038700
00038800
00038900
00039000
00039100
00039200
00039300
00039400
00039500
00039600
00039700
00039800
00039900
00040000
00040100
00040200
00040300
00040400
00040500
00040600
00040700
00040800
00040900
00041000
00041100
00041200
00041300
00041400
00041500
00041600
00041700
00041800
00041900
00042000
00042100
00042200
00042300

```

```

    <INTEGER-FOR-LIST>                                00042400
    / <BOOLEAN-VARIABLE> @S CODEFORVARIABLE <ARROW>  00042500
    <BOOLEAN-FOR-LIST>                                00042600
    / <STRING-VARIABLE> @S CODEFORVARIABLE <ARROW>    00042700
    <STRING-FOR-LIST>                                  00042800
    / <POINTER-VARIABLE> @S CODEFORVARIABLE <ARROW>    00042900
    <POINTER-FOR-LIST>                                00043000
    / <PATTERN-VARIABLE> @S CODEFORVARIABLE <ARROW>    00043100
    <PATTERN-FOR-LIST> ]                               00043200
    #DO @S CODEENDFORLIST <STATEMENT> @S CODEFRANCHBACKFOR 00043300
    / #FOR <SIMPLE-INTEGER-EXPRESSION> #TIMES #DO        00043400
    @S CODEFORCOUNTER <STATEMENT> @S CODEFRANCHBACKFORTIMES 00043500
    / <OPEN-LABEL> <ITERATIVE-STATEMENT> <CLOSE-LABEL> ; 00043600
<INTEGER-FOR-LIST> ::= LIST <FOR-LIST-ELEMENT> SEPARATOR #, ; 00043700
<FOR-EXPRESSION> ::= <INTEGER-EXPRESSION>              00043800
    / #VALUE #( @S CODEBEGINVALUE <INTEGER-EXPRESSION> 00043900
    @S CODEENDVALUE #) ;                                00044000
<FOR-LIST-ELEMENT> ::= <FOR-EXPRESSION> @S CODESINGLEFORASSIGNMENT 00044100
    / <FOR-EXPRESSION> @S CODEINITIALFORASSIGNMENT      00044200
    [ #TO @S CODEBEGINFORLIMIT <FOR-EXPRESSION>        00044300
    @S CODEENDFORLIMIT [ @S CODEDEFAULTFORINCREMENT    00044400
    / #BY @S CODEBEGINFORRY <FOR-EXPRESSION>            00044500
    @S CODEENDFORRY ]                                   00044600
    / [ @S CODENOSTEPFORCONDITIONAL                     00044700
    / #STEP @S CODEBEGINFORSTEP <FOR-EXPRESSION>       00044800
    @S CODEENDFORSTEP ] [ #UNTIL @S CODEBEGINFORUNTIL   00044900
    <BOOLEAN-EXPRESSION> @S CODEENDFORUNTIL             00045000
    / #WHILE @S CODEBEGINFORWHILE <BOOLEAN-EXPRESSION> 00045100
    @S CODEENDFORWHILE ] ] ;                           00045200
<BOOLEAN-FOR-LIST> ::= LIST [ <BOOLEAN-EXPRESSION> @S CODEFORLISTEXP ] 00045300
    SEPARATOR #, ;                                     00045400
<POINTER-FOR-LIST> ::= LIST [ <POINTER-EXPRESSION> @S CODEFORLISTEXP ] 00045500
    SEPARATOR #, ;                                     00045600
<PATTERN-FOR-LIST> ::= LIST [ <PATTERN-EXPRESSION> @S CODEFORLISTEXP ] 00045700
    SEPARATOR #, ;                                     00045800
<STRING-FOR-LIST> ::= LIST [ <STRING-EXPRESSION> @S CODEFORLISTEXP ] 00045900
    SEPARATOR #, ;                                     00046000
<ESCAPE-STATEMENT> ::= #LEAVE <#I> @S LEAVELABEL      00046100
    / #RETURN                                           00046200
    [ @S CHECKTHATPROCEDUREISNOTTYPED                  00046300
    / @T INTEGERPROCEDURE #( <INTEGER-EXPRESSION>      00046400
    @S CODERETURNINTEGER #)                            00046500
    / @T BOOLEANPROCEDURE #( <BOOLEAN-EXPRESSION>      00046600
    @S CODERETURNBOOLEAN #)                            00046700
    / @T STRINGPROCEDURE #( <STRING-EXPRESSION>        00046800
    @S CODERETURNSTRING #)                             00046900
    / @T PATTERNPROCEDURE #( <PATTERN-EXPRESSION>      00047000
    @S CODERETURNPATTERN #)                            00047100
    / @T POINTERPROCEDURE #( <POINTER-EXPRESSION>      00047200
    @S CODERETURNPOINTER #)                            00047300
    / @T STRUCTUREPROCEDURE #( <STRUCTURE-EXPRESSION> #) 00047400
    @S CHECKTHATRETURNSTRUCTUREMATCHESPROCEDURESTRUCTUR ] ; 00047500
<PROCESS-STATEMENT> ::= #SPAWN #( @S CODEBEGINSPAWN    00047600
    <PROCEDURE-STATEMENT> @S CODEENDSPAWN #)            00047700
    / #APPEND #( @S CODEBEGINAPPEND <PROCEDURE-STATEMENT> 00047800
    @S CODEENDAPPEND #)                                00047900
    / #INITIATE #( @S CODEBEGININITIATE <PROCEDURE-STATEMENT> # 00048000
    <PROCESS-VARIABLE> @S CODEENDINITIATE #)           00048100
    / #TERMINATE                                       00048200
    [ @S CODELOMETERMINATE                             00048300
    / #( [ ##8 #, @S CODEDEFAULTTERMINATE @S CODECAUSETERMINATE 00048400

```

```

    <EVENT-LIST> @S CODEENDCAUSETERMINATE                                00048500
  / @S CODEFFGINPROCESSSTERMINATE <PROCESS-VARIABLE>                  00048600
    @S CODEENDPROCESSSTERMINATE                                         00048700
    [ #, CODEFCAUSETERMINATE <EVENT-LIST>                               00048800
      @S CODEFENDCAUSETERMINATE ] &#) ]                                00048900
  / #SUSPEND #( @S CODEBEGINsuspend <PROCESS-VARIABLE>                 00049000
    @S CODEENDSUSPEND #)                                                 00049100
  / #RESTART #( @S CODEBEGINRESTART <PROCESS-VARIABLE>                 00049200
    @S CODEENDRESTART #) ]                                              00049300
<EVENT-LIST> ::= LIST [ <EVENT-VARIABLE> @S CODEEVENTLIST ]          00049400
  SEPARATOR #, ]                                                         00049500
<EVENT-STATEMENT> ::= #CAUSE #( @S CODEBGINCAUSE <EVENT-LIST>         00049600
  @S CODEENDCAUSE #)                                                     00049700
  / #WAIT #( @S CODEBEGINWAIT <EVENT-LIST> @S CODEENDWAIT #) ]        00049800
<PATTERN-MATCHING-STATEMENT> ::= <STRING-VARIABLE>                    00049900
  [ @S CODEFLOATINGMATCH / @S CODEFIXEDMATCH #FIXED ]                 00050000
  <PATTERN-EXPRESSION> @S CODEPATTERNMATCH                              00050100
  [ @S CODENOMATCHASSIGNMENT                                           00050200
    / <ARROW> @S CODEMATCHASSIGNMENT <STRING-EXPRESSION>              00050300
    @S CODEENDMATCHASSIGNMENT ] ]                                       00050400
<INTEGER-ASSIGNMENT> ::= <INTEGER-VARIABLE> @S SETSTAROK <ARROW>      00050500
  <INTEGER-EXPRESSION> @S INTEGERSTORE                                  00050600
  / <INTEGER-FUNCTION-ID> <ARROW>                                         00050700
    <INTEGER-EXPRESSION> @S INTEGERFUNCTIONSTORE;                       00050800
<INTEGER-FUNCTION-ID> ::= <#I> @T TESTINTEGERFUNCTIONID;              00050900
<BOOLEAN-ASSIGNMENT> ::= <BOOLEAN-VARIABLE> @S SETSTAROK <ARROW>     00051000
  <BOOLEAN-EXPRESSION> @S BOOLEANSTORE                                  00051100
  / <BOOLEAN-FUNCTION-ID> <ARROW>                                         00051200
    <BOOLEAN-EXPRESSION> @S BOOLEANFUNCTIONSTORE;                      00051300
<BOOLEAN-FUNCTION-ID> ::= <#I> @T TESTBOOLEANFUNCTIONID;              00051400
<STRING-ASSIGNMENT> ::= <STRING-VARIABLE> @S SETSTAROK <ARROW>        00051500
  <STRING-EXPRESSION> @S STRINGSTORE                                    00051600
  / <STRING-VARIABLE> #, # [ <INTEGER-EXPRESSION>                      00051700
    @S CODEFIELDSTARBYNAME #; <INTEGER-EXPRESSION>                    00051800
    @S CODEFIELDLENGTHBYNAME #] @S SETSTAROK <ARROW>                  00051900
    <STRING-EXPRESSION> @S PARTIALSTRINGSTORE                          00052000
  / <STRING-FUNCTION-ID> <ARROW>                                         00052100
    <STRING-EXPRESSION> @S STRINGFUNCTIONSTORE;                        00052200
<STRING-FUNCTION-ID> ::= <#I> @T TESTSTRINGFUNCTIONID;                 00052300
<POINTER-ASSIGNMENT> ::= <POINTER-VARIABLE> <ARROW>                  00052400
  <POINTER-EXPRESSION> @S POINTERSTORE                                  00052500
  / <POINTER-FUNCTION-ID> <ARROW>                                         00052600
    <POINTER-EXPRESSION> @S POINTERFUNCTIONSTORE;                      00052700
<POINTER-FUNCTION-ID> ::= <#I> @T TESTPOINTERFUNCTIONID;              00052800
<PATTERN-ASSIGNMENT> ::= <PATTERN-VARIABLE> <ARROW>                  00052900
  <PATTERN-EXPRESSION> @S PATTERNSTORE                                  00053000
  / <PATTERN-FUNCTION-ID> <ARROW>                                         00053100
    <PATTERN-EXPRESSION> @S PATTERNFUNCTIONSTORE;                      00053200
<PATTERN-FUNCTION-ID> ::= <#I> @T TESTPATTERNFUNCTIONID;              00053300
<INTEGER-EXPRESSION> ::= <SIMPLE-INTEGER-EXPRESSION>                  00053400
  / <INTEGER-ASSIGNMENT> @S CODENONDESTRUCTINTEGERSTORE;              00053500
<SIMPLE-INTEGER-EXPRESSION> ::= <TERM>                                 00053600
  / <SIMPLE-INTEGER-EXPRESSION> #+ <TERM> @S CODEINTEGERADD            00053700
  / <SIMPLE-INTEGER-EXPRESSION> #- <TERM> @S CODEINTEGERSUBTRACT;      00053800
<TERM> ::= <FACTOR>                                                    00053900
  / <TERM> ** <FACTOR> @S CODEINTEGERMULTIPLY                           00054000
  / <TERM> #/ <FACTOR> @S CODEDIVIDE                                    00054100
  / <TERM> #RDIV <FACTOR> @S CODERDIV                                    00054200
  / <TERM> #CDIV <FACTOR> @S CODECDIV                                    00054300
  / <TERM> #MOD <FACTOR> @S CODEMOD ]                                     00054400
<FACTOR> ::= #+ <PRIMARY> / <PRIMARY> / #- <PRIMARY> @S CODENEGATIVE 00054500

```

```

/ <FACTOR> #POWER [#+] & <PRIMARY> @S CODEPOWER 00054600
/ <FACTOR> #POWER #- <PRIMARY> @S CODENEGATIVE @S CODEPOWER ; 00054700
<PRIMARY> ::= [ <*N> @S CODEINTEGERCONSTANT 00054800
/ ** @T TESTINTEGERSTARALLOWED 00054900
/ <INTEGER-VARIABLE> @S CODEINTEGERVALUECALL 00055000
/ <INTEGER-FUNCTION> @S CODEINTEGERFUNCTION 00055100
/ #IF <BOOLEAN-EXPRESSION> @S CODEEXPSKIPTHENONFALSE 00055200
#THEN <INTEGER-EXPRESSION> @S CODEEXPSKIPFLSF 00055300
@S CODEEXPSKIPTOELSEFOUND 00055400
#ELSE <INTEGER-EXPRESSION> @S CODEEXPSKIPPASTELSEFOUND #FI 00055500
/ #CASE <INTEGER-EXPRESSION> #OF @S CODEEXPCASERANCH 00055600
LIST [ <INTEGER-EXPRESSION> @S CODEEXPCASEENTRY ] 00055700
SEPARATOR #, 00055800
[ #DEFAULT @S CODEEXPCASEDEFAULT <INTEGER-EXPRESSION> 00055900
@S CODEENDEXPCASEDEFAULT / @S CODENOEXPCASEDEFAULT ] 00056000
#ESAC @S CODEENDOFEXPCASE 00056100
/ #ABS #( <INTEGER-EXPRESSION> #) @S CODEABS 00056200
/ #SIGN #( <INTEGER-EXPRESSION> #) @S CODESIGN 00056300
/ #BINARY #( <STRING-EXPRESSION> #) @S CODEINTGFROMSTRING 00056400
/ #LENGTH #( <STRING-EXPRESSION> #) @S CODELENGTHOFSTRING 00056500
/ #CHARSIZE #( <STRING-EXPRESSION> #) @S CODECHARSIZEOFSTRING 00056600
/ #UPPER #( [ <STRUCTURE-FIRST-PART> #. ] & 00056700
<*I> @S CHECKSUBSCRIPTEDFORUPPER [ <SUBSCRIPT> ] & #) 00056800
@S CODEUPPER @S RESETSPT 00056900
/ #LOWER #( [ <STRUCTURE-FIRST-PART> #. ] & 00057000
<*I> @S CHECKSUBSCRIPTEDFORUPPER [ <SUBSCRIPT> ] & #) 00057100
@S CODELOWER @S RESETSPT 00057200
/ #INTEGER #( <STRING-EXPRESSION> #) @S CODEEXTRACTINTEGER 00057300
/ #( <INTEGER-EXPRESSION> #) ] @S STARNOTALLOWED ; 00057400
<BOOLEAN-EXPRESSION> ::= <SIMPLE-BOOLEAN> 00057500
/ <BOOLEAN-ASSIGNMENT> @S CODENONDESTRUCTBOOLEANSTORE 00057600
/ <PATTERN-MATCHING-STATEMENT> ; 00057700
<SIMPLE-BOOLEAN> ::= <IMPLICATION> 00057800
/ <SIMPLE-BOOLEAN> [ #EQV /#=#> ] <IMPLICATION> @S CODEEQV ; 00057900
<IMPLICATION> ::= <BOOLEAN-TERM> 00058000
/ <IMPLICATION> [ #IMP /#=#> ] <BOOLEAN-TERM> @S CODEIMP ; 00058100
<BOOLEAN-TERM> ::= <BOOLEAN-FACTOR> 00058200
/ <BOOLEAN-TERM> 00058300
[ #OR <BOOLEAN-FACTOR> @S CODEOR 00058400
/ #XOR <BOOLEAN-FACTOR> @S CODEXOR 00058500
/ #ORIF @S CODESKIPFACTORIFTRUE <BOOLEAN-FACTOR> 00058600
@S CODEOR @S CODEENDSKIPFACTOR ] ; 00058700
<BOOLEAN-FACTOR> ::= <BOOLEAN-SECONDARY> 00058800
/ <BOOLEAN-FACTOR> 00058900
[ #AND <BOOLEAN-SECONDARY> @S CODEAND 00059000
/ #ANDIF @S CODESKIP2NDARYIFFALSE <BOOLEAN-SECONDARY> 00059100
@S CODEAND @S CODEEND2NDARYSKIP ] ; 00059200
<BOOLEAN-SECONDARY> ::= <BOOLEAN-PRIMARY> 00059300
/ #NOT <BOOLEAN-PRIMARY> @S CODENOT ; 00059400
<BOOLEAN-PRIMARY> ::= [ #TRUE @S CODETRUE / #FALSE @S CODEFALSE 00059500
/ ** @T TESTBOOLEANSTARALLOWED 00059600
/ <BOOLEAN-VARIABLE> @S CODEROOLEANVALUECALL 00059700
/ <BOOLEAN-FUNCTION> @S CODEROOLEANFUNCTION 00059800
/ #BOOL #( <INTEGER-EXPRESSION> #) @S CODEINTGRTOROOL 00059900
/ <RELATION> 00060000
/ #( <BOOLEAN-EXPRESSION> #) 00060100
/ #IF <BOOLEAN-EXPRESSION> @S CODEEXPSKIPTHENONFALSE 00060200
#THEN <BOOLEAN-EXPRESSION> @S CODEEXPSKIPFLSF 00060300
@S CODEEXPSKIPTOELSEFOUND 00060400
#ELSE <BOOLEAN-EXPRESSION> @S CODEEXPSKIPPASTELSEFOUND #FI 00060500
/ #CASE <INTEGER-EXPRESSION> #OF @S CODEEXPCASERANCH 00060600

```

```

LIST [ <BOOLEAN-EXPRESSION> @S CODEEXPCASEENTRY ] 00060700
SEPARATOR #, 00060800
[ #DEFAULT @S CODEEXPCASEDEFAULT <BOOLEAN-EXPRESSION> 00060900
@S CODEENDEXPCASEDEFAULT / @S CODENOEXPCASEDEFAULT ] 00061000
#ESAC @S CODEENDOFEXPCASE 00061100
] @S STARNOTALLOWED ] 00061200
<RELATION> ::= <INTEGER-EXPRESSION> 00061300
LIST [ <RELATIONAL-OPERATOR> EMPTY <INTEGER-EXPRESSION> 00061400
@S CODEINTEGERRELOP ] 00061500
/ <STRING-EXPRESSION> 00061600
LIST [ <RELATIONAL-OPERATOR> EMPTY <STRING-EXPRESSION> 00061700
@S CODESTRINGRELOP ] 00061800
/ <POINTER-EXPRESSION> 00061900
LIST [ <RELATIONAL-OPERATOR> EMPTY <POINTER-EXPRESSION> 00062000
@S CODEPOINTERRELOP ] ; 00062100
<RELATIONAL-OPERATOR> ::= #= @S CODEEQL / ## @S CODENEQ 00062200
/ #< @S CODELSS / #> @S CODEGTR 00062300
/ [#≤ / #<=] @S CODELEQ / [#≥ / #>=] @S CODEGEQ ; 00062400
<PROCEDURE-ID> ::= [ <STRUCTURE-FIRST-PART> #, ]& 00062500
<*I> @T IDISPROCEDURE @S RESETSPT ; 00062600
<STRING-EXPRESSION> ::= <SIMPLE-STRING-EXPRESSION> 00062700
/ <STRING-ASSIGNMENT> @S CODENONDESTRUCTSTRINGSTORE ; 00062800
<SIMPLE-STRING-EXPRESSION> ::= <STRING-TERM> 00062900
/ <SIMPLE-STRING-EXPRESSION> #SOR <STRING-TERM> @S CODESOR 00063000
/ <SIMPLE-STRING-EXPRESSION> #SXOR <STRING-TERM> @S CODESXOR ; 00063100
<STRING-TERM> ::= <STRING-FACTOR> 00063200
/ <STRING-TERM> #SAND <STRING-FACTOR> @S CODESAND ; 00063300
<STRING-FACTOR> ::= <STRING-PPRIMARY> 00063400
/ #NOT <STRING-PRIMARY> @S CODESNOT ; 00063500
<STRING-PRIMARY> ::= [#MT @S CODEMT 00063600
/ #* @T TESTSTRINGSTARALLOWED 00063700
/ <*S> @S CODESTRINGCONSTANT 00063800
/ <STRING-FUNCTION> @S CODESTRINGFUNCTION 00064000
/ #DECIMAL #( <INTEGER-EXPRESSION> #) @S CODEINTGRTODECIMAL 00064100
/ #CONVERT #( <STRING-EXPRESSION> #, <STRING-EXPRESSION> #, 00064200
<STRING-EXPRESSION> [ @S SETCONVERTDEFAULT 00064300
/ #, <INTEGER-EXPRESSION> #) @S, CODECONVERT 00064400
/ #STRING #( <INTEGER-EXPRESSION> [ @S CODESTRINGDEFAULT 00064500
/ #, <INTEGER-EXPRESSION> #) @S CODEINTGRTOSTRING 00064600
/ #( <STRING-EXPRESSION> #) 00064700
/ #IF <BOOLEAN-EXPRESSION> @S CODEEXPSKIPTHENONFALSE 00065000
#THEN <STRING-EXPRESSION> @S CODEEXPSKIPELSE 00065100
@S CODEEXPSKIPTOELSEFOUND 00065200
#ELSE <STRING-EXPRESSION> @S CODEEXPSKIPPASTELSEFOUND #FI 00065300
/ #CASE <INTEGER-EXPRESSION> #OF @S CODEEXPCASEPRANCH 00065400
LIST [ <STRING-EXPRESSION> @S CODEEXPCASEENTRY ] 00065500
SEPARATOR #, 00065600
[ #DEFAULT @S CODEEXPCASEDEFAULT <STRING-EXPRESSION> 00065700
@S CODEENDEXPCASEDEFAULT / @S CODENOEXPCASEDEFAULT ] 00065800
#ESAC @S CODEENDOFEXPCASE 00065900
] @S STARNOTALLOWED ] 00066000
[ #, #[ <INTEGER-EXPRESSION> @S CODFFIELDSTART # ; 00066100
<INTEGER-EXPRESSION> @S CODFFIELDLLENGTH # ] ]& 00066200
/ <STRING-VARIABLE> @S CODESTRINGVALUECALL ; 00066210
<POINTER-EXPRESSION> ::= <POINTER-VARIABLE> @S CODEPOINTERVALUECALL 00066300
/ <POINTER-FUNCTION> @S CODEPOINTERFUNCTION 00066400
/ <POINTER-ASSIGNMENT> @S CODENONDESTRUCTPOINTERSTORE 00066500
/ #NULL @S CODENUIL 00066600
/ #IF <BOOLEAN-EXPRESSION> @S CODEEXPSKIPTHENONFALSE 00066700
#THEN <POINTER-EXPRESSION> @S CODEEXPSKIPFLSE 00066800
@S CODEEXPSKIPTOELSEFOUND 00066900

```

```

#FLSF <POINTER-EXPRESSION> @S CODEEXPSKIPPASTELSEFOUND #FI 00067000
/ #CASE <INTEGER-EXPRESSION> #OF @S CODEFXPCASEBRANCH 00067100
LIST [ <POINTER-EXPRESSION> @S CODEFXPCASEENTRY ] 00067200
SEPARATOR #, 00067300
[ #DEFAULT @S CODEFXPCASEDEFAULT <POINTER-EXPRESSION> 00067400
@S CODEFNDEXPCASEDEFAULT / @S CODENDEXPCASEDEFAULT ] 00067500
#ESAC @S CODEENDOFEXPCASE 00067600
/ #PTR #( @S SCANPTRFUNCTION 00067700
[ <STRUCTURE-FIRST-PART> #. J& <+I> @S VERIFYCANBEPOINTEDTO 00067800
[ @S CODEPTRUNSUBSCRIPTED 00067900
/ <SUBSCRIPT> @S CODEPTRSUBSCRIPTED ] @S RESETSPT #) 00068000
/ #ALLOCATE #( <+I> @S CODEALLOCATEFUNCTION #) ; 00068100
<ALLOCATION-STATEMENT> ::= 00068200
#ALLOCATE #( <+I> @S CODEALLOCATEPROCEDURE #) 00068300
/ #RELEASE #( @S SCANRELEASE [ <STRUCTURE-FIRST-PART> #. J& 00068400
<+I> @S CODERELEASE @S RESETSPT #) ; 00068500
<PATTERN-EXPRESSION> ::= #<#< <PATTERN-ALTERNATIVE-LIST> #>#> 00068600
/ <PATTERN-ASSIGNMENT> @S CODEFNONDESTRUCTPATTERNSTORE 00068700
/ #SPAN #( <STRING-EXPRESSION> [ @S CODESPANDEFAULT 00068800
/ #, <INTEGER-EXPRESSION> ] #) @S CODESPAN 00068900
/ #ANY [ @S CODEANYDEFAULT / #( <INTEGER-EXPRESSION> #) ] 00069000
@S CODEFANY 00069100
/ #EXCEPT #( <STRING-EXPRESSION> [ @S CODEEXCEPTDEFAULT 00069200
/ #, <INTEGER-EXPRESSION> ] #) @S CODEEXCEPT 00069300
/ #IF <BOOLEAN-EXPRESSION> @S CODEEXPSKIPTHENONFALSE 00069400
#THEN <PATTERN-EXPRESSION> @S CODEEXPSKIPELSE 00069500
@S CODEEXPSKIPTDELSEFOUND 00069600
#ELSE <PATTERN-EXPRESSION> @S CODEEXPSKIPPASTELSEFOUND #FI 00069700
/ #CASE <INTEGER-EXPRESSION> #OF @S CODEFXPCASEBRANCH 00069800
LIST [ <PATTERN-EXPRESSION> @S CODEFXPCASEENTRY ] 00069900
SEPARATOR #, 00070000
[ #DEFAULT @S CODEFXPCASEDEFAULT <PATTERN-EXPRESSION> 00070100
@S CODEFNDEXPCASEDEFAULT / @S CODENDEXPCASEDEFAULT ] 00070200
#ESAC @S CODEENDOFEXPCASE ; 00070300
<PATTERN-ALTERNATIVE-LIST> ::= <PATTERN-FACTOR> 00070400
/ <PATTERN-ALTERNATIVE-LIST> #/ <PATTERN-FACTOR> @S CODEPOR ; 00070500
<PATTERN-FACTOR> ::= [ <STRING-VARIABLE> FOLLOWED BY <ARROW> ] & 00070600
LIST [ <STRING-VARIABLE> / <+S> / #MT 00070700
/ <PATTERN-VARIABLE> 00070800
/ <PATTERN-FUNCTION> 00070900
/ <PATTERN-EXPRESSION> ] SEPARATOR #& ; 00071000
<INTEGER-VARIABLE> ::= [ <POINTER-SPECIFIER> ] & <INTEGER-NAME> 00071100
@S CODEINTEGERVARIABLE 00071200
/ <ACCESS-VARIABLE> @T ACCESSINTEGER ; 00071300
<BOOLEAN-VARIABLE> ::= [ <POINTER-SPECIFIER> ] & <BOOLEAN-NAME> 00071400
@S CODEBOOLEANVARIABLE 00071500
/ <ACCESS-VARIABLE> @T ACCESSBOOLEAN ; 00071600
<STRING-VARIABLE> ::= [ <POINTER-SPECIFIER> ] & <STRING-NAME> 00071700
@S CODESTRINGVARIABLE 00071800
/ <ACCESS-VARIABLE> @T ACCESSSTRING ; 00071900
<POINTER-VARIABLE> ::= [ <POINTER-SPECIFIER> ] & <POINTER-NAME> 00072000
@S CODEPOINTERVARIABLE 00072100
/ <ACCESS-VARIABLE> @T ACCESSPOINTER ; 00072200
<PATTERN-VARIABLE> ::= [ <POINTER-SPECIFIER> ] & <PATTERN-NAME> 00072300
@S CODEPATTERNVARIABLE 00072310
/ <ACCESS-VARIABLE> @T ACCESSPATTERN ; 00072400
<ACCESS-VARIABLE> ::= <+I> @T TESTACCESS 00072500
[ @S MUSTREUNSUBSCRIPTED / <SUBSCRIPT> 00072600
@S CHECKSUBSCRIPTMATCH ] ; 00072700
<PROCESS-VARIABLE> ::= [ <POINTER-SPECIFIER> ] & <PROCESS-NAME> 00072800
/ <ACCESS-VARIABLE> @T ACCESSPROCESS ; 00072900

```

<CODE-VARIABLE> ::= [<POINTER-SPECIFIER>] & <CODE-NAME>	00073000
/ <ACCESS-VARIABLE> @T ACCESSCODE ;	00073100
<FILE-VARIABLE> ::= [<POINTER-SPECIFIER>] & <FILE-NAME>	00073200
/ <ACCESS-VARIABLE> @T ACCESSFILE ;	00073300
<STRUCTURE-VARIABLE> ::= [<POINTER-SPECIFIER>] & <STRUCTURE-FIRST-PART>	00073400
/ <ACCESS-VARIABLE> @T ACCESSSTRUCTURE ;	00073500

OSL2 /SMANTIX LISTED BY THE SIS ON JANUARY 3, 1971 AT 1:21 A.M.

```

* NOLIST SAVEZIPDECK DISK TEST                                00000010
$BEGIN $UPPERN(200);                                           00000020
  INTEGER  IDCOUNTER,                                           % THIS IS THE CURRENT NUMBER OF 00000100
                                                                % DECLARATION IDS PROCESSED IN 00000200
                                                                % THE CURRENT BLOCK HEAD.      00000300
  IDLISTCOUNTER,                                                % THIS CONTAINS THE CURRENT      00000400
                                                                % NUMBER OF IDS PROCESSED IN AN 00000500
                                                                % ID PART.                     00000600
  CURRENTDECLTYPE,                                              % THIS IS THE DEFAULT TYPE FOR 00000700
                                                                % THE CURRENT DECLARATION .    00000800
  CURRENTDECLDOWN,                                              % =1 IF THIS DECLARATION IS     00000900
                                                                % OWN.                         00001000
  CURRENTDECLNODE,                                              % =1 IF THIS DECLARATION IS     00001100
                                                                % NODE.                       00001200
  CURRENTDECLMACRO,                                             % =1 IF THIS DECLARATION IS     00001300
                                                                % MACRO.                      00001400
  CURRENTNUMBEROF SUBS,                                         % WHEN A BOUND PAIR LIST IS     00001500
                                                                % BEING SCANNED IN THE PARSE OF 00001600
                                                                % A SUBSCRIPT LIST OR         00001700
                                                                % AN ARRAY PART, THIS VARIABLE 00001800
                                                                % WILL CONTAIN THE NUMBER OF   00001900
                                                                % SUBSCRIPTS ALREADY RECOGNIZED. 00002000
  CURRENTPROCEDUREID,                                           % THIS IS A POINTER TO THE      00002100
                                                                % CURRENTLY BEING DEFINED      00002200
                                                                % PROCEDURE IN IDSTACK.       00002300
  CURRENTEXPRESSIONTYPE,                                         % THIS VARIABLE                 00002400
                                                                % CONTAINS THE TYPE OF THE LAST 00002500
                                                                % EXPRESSION SCANNED.         00002600
  CURRENTSTRUCTUREPTR,                                          % IF A STRUCTURE VARIABLE WAS   00002700
                                                                % LAST SCANNED, THIS          00002800
                                                                % CONTAINS A POINTER INTO     00002900
                                                                % STRUCTURETAB TO THE APPROP- 00003000
                                                                % RIATE SUBSTRUCTURE VECTOR.  00003100
  CURRENTNUMBEROF SUBSTRUCTURES,                                % THIS IS THE LENGTH OF        00003300
                                                                % THE ABOVE POINTED AT VECTOR. 00003400
  CURRENTIDPTR,                                                 % IF THE LAST ID SCANNED WAS   00003500
                                                                % PART OF A STRUCTURE, THEN THIS 00003600
                                                                % VARIABLE POINTS INTO        00003700
                                                                % STRUCTURETAB WHERE THE ID IS  00003800
                                                                % TABULATED.                  00003900
  CURRENTFORMALPARAMETERLISTPTR,                                % THIS VARIABLE IS USED        00004100
                                                                % DURING ACTUAL PARAMETER SCANS 00004200
                                                                % TO POINT TO THE BEGINNING OF 00004300
                                                                % THE DECLARED FORMAL PARAMETER 00004400
                                                                % LIST IN STRUCTURETAB.       00004500
  CURRENTNUMBEROFFORMALPARAMETERS,                              % THIS IS THE LENGTH OF        00004700
                                                                % THE ABOVE VECTOR.           00004800
  CURRENTACTUALPARAMETERCOUNT,                                % THIS IS THE POSITION IN THE    00005000
                                                                % ACTUAL PARAMETER LIST OF     00005100
                                                                % THE ACTUAL PARAMETER CURRENTLY 00005200
                                                                % BEING SCANNED.              00005300
  CURRENTMAXACCESSNUMBER,                                       % THIS VARIABLE IS THE CODE     00005500

```

		* NUMBER OF THE LAST ASSIGNED	00005600
		* ACCESS TYPE, E.G. FIFO	00005700
		* ACCESS MIGHT BE #1 AND LIFO	00005800
		* ACCESS MIGHT BE #2. IF THAT	00005900
		* IS ALL THERE ARE, THEN	00006000
		* CURRENTMAXACCESSNUMBER	00006100
		* WOULD BE 2.	00006200
CURRENTSEGMENT,		* CURRENT CODE SEGMENT	00006300
CURRENTADDRESS,		* CURRET CODE ADDRESS	00006400
CURRENTLEVEL,		* CURRENT LEVEL REGISTER	00006500
		* IN USE	00006600
MAXIMUMSEGMENT,		* THE LARGEST SFGMENT	00006700
		* NUMBER COMPILED SO FAR	00006800
CURRENTMAXIMUMDISPLACEMENT,			00006900
		* THE NEXT DISPLACEMENT TO	00007000
		* BE ASSIGNED AT THIS LEVEL	00007100
CURRENTCASECOUNT,		* THIS CONTAINS THE CURRENT	00007200
		* NUMBER OF EXPPESSIONS OR	00007300
		* STATEMENTS SCANNED WITHIN A	00007400
		* CASE EXPRESSION OR CASE	00007500
		* STATEMENT.	00007600
LASTTYPE,		* USED TO HANDLE THE	00007700
		* ASTERISK CONVENTION -	00007800
		* LASTTYPE IS THE TYPE OF	00007900
		* THE LAST VARIABLE SCANNED.	00008000
RELOP,		* RELOP TELLS WHICH	00008100
		* RELATIONAL OPERATOR WAS	00008200
		* LAST SCANNED.	00008300
		* 0 NONE	00008400
		* 1 <	00008500
		* 2 ≤	00008600
		* 3 =	00008700
		* 4 ≥	00008800
		* 5 >	00008900
		* 6 ≠	00009000
CURRENTACCESSID,		* THIS POINTS TO THE	00009100
		* CURRENTLY BEING DEFINED	00009200
POINTER PMESOUT,		* ACCESS TYPE ENTRY IN IDSTACK.	00009300
		* THIS POINTER IS INITIALIZED	00009400
		* IN ACTION(0) TO POINT TO THE	00009500
		* ARRAY MESOUT.	00009600
POOLEAN STRUCTUREISOWN,		* THIS SWITCH IS TRUE IF AN	00009700
		* OWN STRUCTURE IS BEING	00009800
		* DECLARED.	00009900
PRINTOBJECT,		* TRUE IF OBJECT LISTING	00010000
		* IS DESIRED	00010100
SCANNINGFORMALPARAMETERDECLARATIONS,			00010200
		* THIS SWITCH IS TRUE IF A	00010300
		* FORMAL PARAMETER DECLARATION	00010400
		* IS BEING SCANNED. IT MUST BE	00010500
		* PUSHED WHEN A PROCEDURE	00010600
		* DECLARATION IS ENCOUNTERED FOR	00010700
		* A FORMAL PARAMETER.	00010800
ASTERISKALLOWED,		* THIS IS TRUE IF THE	00010900
		* ASTERISK CONVENTION MAY BE	00011000
		* INVOKED IN AN ASSIGNMENT	00011100
		* STATEMENT.	00011200
NONAMECALL,		* TRUE IF A STRUCTURE OR	00011300
		* A SUBSCRIPT HAVE BEEN	00011400
		* SCANNED AND NO NAME CALL	00011500
		* SHOULD BE EMITTED BY THE	00011600

```

% VARIABLE CODE PROCEDURES. 00011700
CURRENTSTRUCTUREPTRISINIDSTACK; 00011800
% THIS VARIABLE IS TRUE IF 00011900
% CURRENTSTRUCTUREPTR POINTS 00012000
% INTO THE IDSTACK INSTEAD OF 00012100
% STRUCTURETAB. 00012200
ARRAY MESOUT[0:17], % THIS ARRAY IS USED TO STORE 00012300
% OUTPUT MESSAGES FOR THE SOURCE 00012400
% LISTING. THE POINTER PMESOUT 00012500
% WILL ALWAYS POINT TO MESOUT[0] 00012600
ACCESSTARLE[0:63]; % THIS ARRAY CONTAINS 00012700
% POINTERS INTO THE IDSTACK 00012800
% FOR THE LOCATION OF EACH 00012900
% ACCESS TYPE DECLARED. 00013000
DEFINE BLANKMESOUT = REPLACE PMESOUT BY " " FOR 17 WORDS #, 00013100
% THIS STATEMENT WILL PUT 00013200
% BLANKS INTO THE ARRAY MESOUT. 00013300
EPRMS = TSKTSK(FALSE,FALSE);BLANKMESOUT; 00013400
REPLACE PMESOUT BY "**** ", #, 00013500
% THIS DEFINE WILL PUT THE 00013600
% FIRST PART OF AN ERROR MESSAGE 00013700
% INTO THE ARRAY MESOUT. 00013800
WRITEMESOUT = WRITE(LINE,17,MESOUT[*]) #, 00013900
% THIS DEFINE WILL WRITE THE 00014000
% ARRAY MESOUT INTO THE FILE 00014100
% LINE. 00014200
UNDECL = 0 #, % THIS ID IS UNDECLARED. 00014300
INT = 1 #, % THIS ID IS INTEGER. 00014400
BOOL = 2 #, % THIS ID IS BOOLEAN. 00014500
STR = 3 #, % THIS ID IS STRING. 00014600
PTR = 4 #, % THIS ID IS POINTER. 00014700
EVNT = 5 #, % THIS ID IS EVENT. 00014800
PROS = 6 #, % THIS ID IS PROCESS. 00014900
PATT = 7 #, % THIS ID IS PATTERN. 00015000
STRUCT = 8 #, % THIS ID IS STRUCTURE. 00015100
ACCESS = 9 #, % THIS ID IS ACCESS. 00015200
OSLCODE = 10 #, % THIS ID IS CODE. 00015300
OSLFILE = 11 #, % THIS ID IS FILE. 00015400
LARL = 12 #, % THIS ID IS A LARL 00015500
ACCESSVAR= 13 #, % THIS ID IS AN ACCESS VARIABLE 00015600
LEAVE= GO TO #, % THIS ALLOWS ONE TO SIMULATE 00015700
% AN OSL LEAVE. 00015800
REGINS = BEGIN LARL FRUDECORP; #, 00015900
ENDS = ;ERUDECORP; END #, 00016000
% REGINS AND ENDS WHEN USED 00016100
% IN PLACE OF THE OUTERMOST 00016200
% BEGIN AND END IN A PROCEDURE 00016300
% DECLARATION BODY ALLOW ONE TO 00016400
% USE AN OSL RETURN STATEMENT TO 00016500
% RETURN FROM THE PROCEDURE. 00016600
THENS = THEN BEGIN #, 00016700
ELSES = END ELSE BEGIN #, 00016800
FI = END #, 00016900
% THE ABOVE THREE DEFINES 00017000
% ALLOW ONE TO USE OSL2 IF 00017100
% STATEMENTS OF THE FORM: 00017200
% IF <B.F.> 00017300
% THENS 00017400
% <STATEMENT>; 00017500
% <STATEMENT>; 00017600
% 00017700

```

```

      X      .      00017800
      X      :      00017900
      X      :      00018000
      X      <STATEMENT>; 00018100
      X      <STATEMENT> 00018200
      X      ELSE$      00018300
      X      <STATEMENT>; 00018400
      X      <STATEMENT>; 00018500
      X      :      00018600
      X      :      00018700
      X      :      00018800
      X      :      00018900
      X      <STATEMENT>; 00019000
      X      <STATEMENT> 00019100
      X      FI      00019200
BRANCH(BRANCH1,BRANCH2,BRANCH3) = 00019300
BEGIN      00019400
    IF PRINTOBJECT      00019500
    THEN      00019600
    BEGIN      00019700
        BLANKMESOUT;      00019800
        REPLACE PMESOUT BY CURRENTSEGMENT FOR 3 DIGITS, 00019900
        "!",BRANCH1 FOR 4 DIGITS, "*** ",BRANCH3; 00020000
        REPLACE FMESOUT+35 BY "(", 00020100
        CURRENTSEGMENT FOR 3 DIGITS,"!", 00020200
        BRANCH2 FOR 4 DIGITS,"")"; 00020300
        WRITEMESOUT      00020400
    END      00020500
END #,      00020600
      X      THIS DEFINE OUTPUTS A      00020700
      X      BRANCH INSTRUCTION OF THE      00020800
      X      FORM:      00020900
      X      SEG:ADDR BRANCH (SEG:ADDR) 00021000
OPERATOR(OPERATOR1) =      00021100
BEGIN      00021200
    IF PRINTOBJECT      00021300
    THEN      00021400
    BEGIN      00021500
        BLANKMESOUT;      00021600
        REPLACE PMESOUT BY CURRENTSEGMENT FOR 3 DIGITS, 00021700
        "!",CURRENTADDRESS FOR 4 DIGITS, 00021800
        " ",OPERATOR1;      00021900
        WRITFMESOUT      00022000
    END;      00022100
    $INCR(CURRENTADDRESS) 00022200
END #,      00022300
      X      THIS DEFINE EMITS AN      00022400
      X      OPERATOR AS OBJECT CODE.      00022500
      X      CURRENTADDRESS IS INCREMENTED. 00022600
XOPERATOR(XOPERATOR1) =      00022700
BEGIN      00022800
    IF PRINTOBJECT      00022900
    THEN      00023000
    BEGIN      00023100
        BLANKMESOUT;      00023200
        REPLACE PMESOUT BY CURRENTSEGMENT FOR 3 DIGITS, 00023300
        "!",CURRENTADDRESS FOR 4 DIGITS, 00023400
        "*** ",XOPERATOR1;      00023500
        WRITFMESOUT      00023600
    END;      00023700
    $INCR(CURRENTADDRESS) 00023800

```

```

END #,
                                * THIS DEFINE EMITS AN 00023900
                                * OPERATOR AS OBJECT CODE. 00024000
                                * CURRENTADDRESS IS INCREMENTED. 00024100
                                * THE OPERATOR IS MARKED AS 00024200
                                * AN OVERWRITTEN INSTRUCTION. 00024300
                                * AN OVERWRITTEN INSTRUCTION. 00024400
COUPLE(COUPLE1,COUPLE2,COUPLE3,COUPLE4) = 00024500
BEGIN 00024600
    IF PRINTOBJECT 00024700
    THEN 00024800
    BEGIN 00024900
        BLANKMESOUT; 00025000
        REPLACE PMESOUT BY CURRENTSEGMENT FOR 3 DIGITS, 00025100
        ":",CURRENTADDRESS FOR 4 DIGITS, 00025200
        " ",COUPLE1," (", 00025300
        COUPLE2 FOR 2 DIGITS," ", 00025400
        COUPLE3 FOR 4 DIGITS,") ", 00025500
        COUPLE4; 00025600
        WRITEMESOUT 00025700
    END; 00025800
    $INCR(CURRENTADDRESS) 00025900
END #, 00026000
                                * THIS DEFINE EMITS AN 00026100
                                * OPERATION CODE FOLLOWED BY 00026200
                                * A (LEVEL,DISPLACEMENT) COUPLE. 00026300
OUTPUTRELOP(OUTPUTRELOP1) = 00026400
BEGIN 00026500
    IF PRINTOBJECT 00026600
    THEN 00026700
    BEGIN 00026800
        INTEGER I; 00026900
        POINTER P; 00027000
        GETLKSVM(1); 00027100
        BLANKMESOUT; 00027200
        REPLACE P:PMESOUT BY 00027300
            CURRENTSEGMENT FOR 3 DIGITS, 00027400
            ":",CURRENTADDRESS FOR 4 DIGITS, 00027500
            " ",OUTPUTRELOP1,"-"; 00027600
        CASE RELOP OF 00027700
        BEGIN 00027800
            ; 00027900
            REPLACE P:P BY "LESS-THAN"; 00028000
            REPLACE P:P BY "LESS-OR-EQUAL"; 00028100
            REPLACE P:P BY "EQUAL"; 00028200
            REPLACE P:P BY "GREATER-OR-EQUAL"; 00028300
            REPLACE P:P BY "GREATER-THAN"; 00028400
            REPLACE P:P BY "NOT-EQUAL" 00028500
        END; 00028600
        IF (I+$OPD(MSTACK,$TOP+1,PSYM)) = "<" 00028700
        OR I = "<=" OR I = "=" OR I = ">=" 00028800
        OR I = ">" OR I = "<=" 00028900
        THEN REPLACE P:P BY "-PARTIAL-DESTRUCTIVE"; 00029000
        WRITEMESOUT; 00029100
    END; 00029200
    RELOP + 0; 00029300
END #, 00029400
                                * THIS DEFINE EMITS 00029500
                                * RELATION OPERATOR CODE 00029600
                                * FOR INTEGERS,STRINGS, 00029700
                                * AND POINTERS. 00029800
OPFRFIRSTPART = 00029900

```

```

BEGIN                                00030000
  IF PRINTOBJECT                     00030100
  THEN                               00030200
  BEGIN                              00030300
    BLANKMESOUT;                     00030400
    REPLACE PMFSOUT BY CURRENTSEGMENT FOR 3 DIGITS, 00030500
    ":", CURRENTADDRESS FOR 4 DIGITS, 00030600
    " " #,                           00030700
    * THIS DEFINE PUTS OUT T         00030800
    * THE FIRST PART OF AN OPERATOR. 00030900
  OPERLASTPART=                      00031000
    WRITMESOUT                       00031100
  END;                               00031200
  $INCR(CURRENTADDRESS)              00031300
END #,                               00031400
                                     00031500
    * THIS DEFINE OUTPUTS
    * THE LAST PART OF AN OPERATOR 00031600
    * IF PRINTOBJECT IS TURNED ON. 00031700
DROP = BEGIN DEFINE 0137=XX#;#;     00031800
    * THIS DEFINE IS USED TO DROP 00031900
    * INTO A NEW SEGMENT.           00032000
    * THIS STACK STORES ALL         00100100
    * CURRENTLY DEFINED IDENTIFIERS 00100200
    * AND THE INFORMATION RELATIVE 00100300
    * TO EACH OF THEM.              00100400
    * THIS FIELD HOLDS THE DSL      00100500
    * TYPE OF THE DATA.             00100600
    * THIS FIELD CONTAINS THE       00100700
    * ACCESS TYPE NUMBER OF DATA   00100800
    * DECLARED ACCESS.              00100900
    * TYPE OF STORE FOR ACCESS      00101000
    * DATA.                         00101100
    * TYPE OF FETCH FOR ACCESS      00101200
    * DATA.                         00101300
    * THIS FIELD CONTAINS THE       00101400
    * NUMBER OF SUBSCRIPTS.          00101500
    * NUMBER OF SUBSTRUCTURES.       00101600
    * NUMBER OF PROCEDURE PARMS.     00101700
    * THE FOLLOWING TYPE FLAGS ARE 00101800
    * STORED HERE:                   00101900
    * MACRO FLAG                     00102000
    * SUBSCRIPTED VARIABLE           00102100
    * PROCEDURE NAME                 00102200
    * NODE FLAG                      00102300
    * DWN FLAG                       00102400
    * MISCELLANEOUS FLAGS            00102500
    * FORMAL PARAMETER               00102600
    * CALL BY VALUE FORMAL PARAM.    00102700
    * THIS FORMAL PARAMETER HAS      00102800
    * BEEN DECLARED IN THE           00102900
    * PROCEDURE HEADING.             00103000
    * THIS IS AN ACCESS PROCEDURE   00103100
    * THIS IS A DUMMY FORMAL         00103200
    * PARAMETER INSERTED INTO AN     00103300
    * ACCESS PROCEDURE IN ORDER      00103400
    * TO INSURE THAT THE ACCESS      00103500
    * VARIABLE IS INCLUDED IN THE    00103600
    * ACCESS PROCEDURE CALL.         00103700
    * THIS POINTS BACK INTO BIGTAB 00103800
    * TO THE ID CHARACTERS AND       00103900
    * CURRENT SEMANTIC FIELD FOR     00104000

```

STABLE(IDSTACK, 1000,

 TYPE = 114,
 ACESSTYPE= 516,
 STORETYPE= 1114,
 FETCHTYPE= 1514,
 NUMBROFSUBS= 1918,
 NUMBRSUBSTRUCT=2718,
 NUMBROF PARMS= 3518,
 TYPEFLAGS= 4315,
 MACRO= 4311,
 SUBSCRIPTED= 4411,
 PROCEDURE= 4511,
 NODE= 4611,
 DWN= 4711,
 MISCLFLAGS= 4918,
 FORMALPARM= 4911,
 VALUEPARM= 5011,
 HASBEENSPEC= 5111,
 ACCESSPROCEDURE=5211,
 DUMMYPARM= 5311,
 BIGTABPTR= 57113,

STRUCTUREPTR= 70:13,	* THIS ID.	00104100
	* THIS POINTS TO SUBSTRUCTURE	00104200
	* DATA IN STRUCTURETAB.	00104300
	* THE SUBSTRUCTURE DATA	00104400
	* BEGINS AT A HIGH ADDRESS	00104500
	* AND CONTINUES TO A LOW ADDRESS	00104600
FORMALPARMPTR= 83:13,	* THIS POINTS TO A FORMAL	00104700
	* PARAMETER LIST IN STRUCTURETAB	00104800
	* THE FORMAL PARAMETER DATA	00104900
	* BEGINS AT A HIGH ADDRESS	00105000
	* AND CONTINUES TO A LOW ADDRESS	00105100
LEVEL = 97:16,	* THIS VARIABLE IS ADDRESSED	00105200
	* AT THIS LEVEL	00105300
FETCHPROCEDURE=103:13,	* THIS POINTS TO AN ACCESS	00105400
	* FETCH PROCEDURE.	00105500
STOREPROCEDURE=116:13,	* THIS POINTS TO AN ACCESS	00105600
	* STORE PROCEDURE.	00105700
OLDSEMFIELD= 129:15,	* THIS CONTAINS THE CONTENTS	00105800
	* OF THE SEM FIELD OF BIGTAB	00105900
	* BEFORE THIS ID WAS DECLARED.	00106000
ISDEFINED= 129:1,	* THIS IS THE BREAKDOWN OF THE	00106100
	* SEM FIELD.	00106200
ISBEINGDEFINED=130:1,	* *	00106300
FORWARDDEFINED=131:1,	* *	00106400
IDSTACKPTR= 132:12,	* *	00106500
DISPLACEMENT = 145:8,	* THIS VARIABLE IS ADDRESSED	00106600
	* AT THIS DISPLACEMENT	00106700
STRUCTPTRBEFORE=153:13,	* THIS IS STRUCTURETAB POINTER	00106800
	* BEFORE THIS MESS WAS DECLARED.	00106900
FETCHAUXPTR= 166:13,	* THIS FIELD IS USED FOR	00107000
	* AUXILLIARY FETCH DATA IN	00107100
	* ACCESS TYPES AND ACCESS	00107200
	* VARIABLES.	00107300
STOREAUXPTR= 179:13,	* AUXILLIARY STORE DATA.	00107400
LABELSEGMENT = 153:15,	* THESE THREE FIELDS	00107500
LABELSTARTADDRESS=166:13,	* DESCRIBE THE BOUNDS OF A	00107600
LABELENDADDRESS=179:13,	* LABEL	00107700
STRUCTURETAB, 1000,	* THIS TABLE-STACK LOOKS	00107800
	* IDENTICAL TO IDSTACK. IT IS	00107900
	* USED TO STORE ALL SUBSTRUCTURE	00108000
	* IDS AND THE INFORMATION	00108100
	* RELATIVE TO THEM.	00108200
TYPE = 1:4,	* THIS FIELD HOLDS THE OSL	00108300
	* TYPE OF THE DATA.	00108400
ACCESSTYPE= 5:6,	* THIS FIELD CONTAINS THE	00108500
	* ACCESS TYPE NUMBER OF DATA	00108600
	* DECLARED ACCESS.	00108700
STORETYPE= 11:4,	* TYPE OF STORE FOR ACCESS	00108800
	* DATA.	00108900
FETCHTYPE= 15:4,	* TYPE OF FETCH FOR ACCESS	00109000
	* DATA.	00109100
NUMBROFSUBS= 19:8,	* THIS FIELD CONTAINS THE	00109200
	* NUMBER OF SUBSCRIPTS.	00109300
NUMBRSUBSTRUCT=27:8,	* NUMBER OF SUBSTRUCTURES.	00109400
NUMBROFPARMS= 35:8,	* NUMBER OF PROCEDURE PARMS.	00109500
TYPEFLAGS= 43:5,	* THE FOLLOWING TYPE FLAGS ARE	00109600
	* STORED HERE:	00109700
MACRO= 43:1,	* MACRO FLAG	00109800
SUBSCRIPTED= 44:1,	* SUBSCRIPTED VARIABLE	00109900
PROCEDURE= 45:1,	* PROCEDURE NAME	00110000
NODE= 46:1,	* NODE FLAG	00110100

OWN=	47:1,	%	OWN FLAG	00110200
MISCFLAGS=	49:8,	%	MISCELLANEOUS FLAGS	00110300
FORMALPARAM=	49:1,	%	FORMAL PARAMETER	00110400
VALUEPARAM=	50:1,	%	CALL BY VALUE FORMAL PARAM.	00110500
HASBEENSPEC=	51:1,	%	THIS FORMAL PARAMETER HAS	00110600
		%	BEEN DECLARED IN THE	00110700
		%	PROCEDURE HEADING.	00110800
ACCESSPROCEDURE=	52:1,	%	THIS IS AN ACCESS PROCEDURE	00110900
DUMMYPARAM=	53:1,	%	THIS IS A DUMMY FORMAL	00111000
		%	PARAMETER INSERTED INTO AN	00111100
		%	ACCESS PROCEDURE IN ORDER	00111200
		%	TO INSURE THAT THE ACCESS	00111300
		%	VARIABLE IS INCLUDED IN THE	00111400
		%	ACCESS PROCEDURE CALL.	00111500
RIGTABPTR=	57:13,	%	THIS POINTS BACK INTO BIGTAB	00111600
		%	TO THE ID CHARACTERS AND	00111700
		%	CURRENT SEMANTIC FIELD FOR	00111800
		%	THIS ID.	00111900
STRUCTUREPTR=	70:13,	%	THIS POINTS TO SUBSTRUCTURE	00112000
		%	DATA IN STRUCTURETAB.	00112100
		%	THE SUBSTRUCTURE DATA	00112200
		%	BEGINS AT A HIGH ADDRESS	00112300
		%	AND CONTINUES TO A LOW ADDRESS	00112400
FORMALPARAMPTR=	83:13,	%	THIS POINTS TO A FORMAL	00112500
		%	PARAMETER LIST IN STRUCTURETAB	00112600
		%	THE FORMAL PARAMETER DATA	00112700
		%	BEGINS AT A HIGH ADDRESS	00112800
		%	AND CONTINUES TO A LOW ADDRESS	00112900
LEVEL =	97:6,	%	THIS VARIABLE IS ADDRESSED	00113000
		%	AT THIS LEVEL	00113100
FETCHPROCEDURE=	103:13,	%	THIS POINTS TO AN ACCESS	00113200
		%	FETCH PROCEDURE.	00113300
STOREPROCEDURE=	116:13,	%	THIS POINTS TO AN ACCESS	00113400
		%	STORE PROCEDURE.	00113500
OLDSEMFIELD=	129:15,	%	THIS CONTAINS THE CONTENTS	00113600
		%	OF THE SEM FIELD OF RIGTAB	00113700
		%	BEFORE THIS ID WAS DECLARED.	00113800
ISDEFINED=	129:1,	%	THIS IS THE BREAKDOWN OF THE	00113900
		%	SEM FIELD. *	00114000
ISBEINGDEFINED=	130:1,	%	*	00114100
FORWARDDEFINED=	131:1,	%	*	00114200
IDSTACKPTR=	132:12,	%	*	00114300
DISPLACEMENT =	145:8,	%	THIS VARIABLE IS ADDRESSED	00114400
		%	AT THIS DISPLACEMENT	00114500
STRUCTPTRBEFORE=	153:13,	%	THIS IS STRUCTURETAB POINTER	00114600
		%	BEFORE THIS MESS WAS DECLARED.	00114700
FETCHAUXPTR=	166:13,	%	THIS FIELD IS USED FOR	00114800
		%	AUXILIARY FETCH DATA IN	00114900
		%	ACCESS TYPES AND ACCESS	00115000
		%	VARIABLES.	00115100
STOREAUXPTR=	179:13,	%	AUXILIARY STORE DATA.	00115200
LABELSEGMENT =	153:15,	%	THESE THREE FIELDS	00115300
LABELSTARTADDRESS=	166:13,	%	DESCRIBE THE BOUNDS OF A	00115400
LABELENDADDRESS=	179:13,	%	LABEL	00115500
\$CELLIDENTITY,				00115600
TYPE =	1:4,	%	THIS FIELD HOLDS THE OSL	00115700
		%	TYPE OF THE DATA.	00115800
ACCESSTYPE=	5:6,	%	THIS FIELD CONTAINS THE	00115900
		%	ACCESS TYPE NUMBER OF DATA	00116000
		%	DECLARED ACCESS.	00116100
STORETYPE=	11:4,	%	TYPE OF STORE FOR ACCESS	00116200

FETCHTYPE=	15:4,	% DATA.	00116300
		% TYPE OF FETCH FOR ACCESS	00116400
NUMBROFSURS=	19:8,	% DATA.	00116500
		% THIS FIELD CONTAINS THE	00116600
NUMBRSUBSTRUCT=	27:8,	% NUMBER OF SUBSCRIPTS.	00116700
NUMBROFPARMS=	35:8,	% NUMBER OF SUBSTRUCTURES.	00116800
TYPEFLAGS=	43:5,	% NUMBER OF PROCEDURE PARMS.	00116900
		% THE FOLLOWING TYPE FLAGS ARE	00117000
		% STORED HERE:	00117100
MACRO=	43:1,	% MACRO FLAG	00117200
SUBSCRIPTED=	44:1,	% SUBSCRIPTED VARIABLE	00117300
PROCEDURE=	45:1,	% PROCEDURE NAME	00117400
NODE=	46:1,	% NODE FLAG	00117500
OWN=	47:1,	% OWN FLAG	00117600
MISCFLAGS=	49:8,	% MISCELLANEOUS FLAGS	00117700
FORMALPARM=	49:1,	% FORMAL PARAMETER	00117800
VALUEPARM=	50:1,	% CALL BY VALUE FORMAL PARM.	00117900
HASBEENSPEC=	51:1,	% THIS FORMAL PARAMETER HAS	00118000
		% BEEN DECLARED IN THE	00118100
		% PROCEDURE HEADING.	00118200
ACCESSPROCEDURE=	52:1,	% THIS IS AN ACCESS PROCEDURE	00118300
DUMMYPARM=	53:1,	% THIS IS A DUMMY FORMAL	00118400
		% PARAMETER INSERTED INTO AN	00118500
		% ACCESS PROCEDURE IN ORDER	00118600
		% TO INSURE THAT THE ACCESS	00118700
		% VARIABLE IS INCLUDED IN THE	00118800
		% ACCESS PROCEDURE CALL.	00118900
BIGTABPTR=	57:13,	% THIS POINTS BACK INTO BIGTAB	00119000
		% TO THE ID CHARACTERS AND	00119100
		% CURRENT SEMANTIC FIELD FOR	00119200
		% THIS ID.	00119300
STRUCTUREPTR=	70:13,	% THIS POINTS TO SUBSTRUCTURE	00119400
		% DATA IN STRUCTURETAB.	00119500
		% THE SUBSTRUCTURE DATA	00119600
		% BEGINS AT A HIGH ADDRESS	00119700
		% AND CONTINUES TO A LOW ADDRESS	00119800
FORMALPARMPTR=	83:13,	% THIS POINTS TO A FORMAL	00119900
		% PARAMETER LIST IN STRUCTURETAB	00120000
		% THE FORMAL PARAMETER DATA	00120100
		% BEGINS AT A HIGH ADDRESS	00120200
		% AND CONTINUES TO A LOW ADDRESS	00120300
LEVEL =	97:6,	% THIS VARIABLE IS ADDRESSED	00120400
		% AT THIS LEVEL	00120500
FETCHPROCEDURE=	103:13,	% THIS POINTS TO AN ACCESS	00120600
		% FETCH PROCEDURE.	00120700
STOREPROCEDURE=	116:13,	% THIS POINTS TO AN ACCESS	00120800
		% STORE PROCEDURE.	00120900
OLDSEMFELD=	129:15,	% THIS CONTAINS THE CONTENTS	00121000
		% OF THE SEM FIELD OF BIGTAB	00121100
		% BEFORE THIS ID WAS DECLARED.	00121200
ISDEFINED=	129:1,	% THIS IS THE BREAKDOWN OF THE	00121300
		% SEM FIELD. *	00121400
ISREINGDEFINED=	130:1,	% *	00121500
FORWARDDDEFINED=	131:1,	% *	00121600
IDSTACKPTR=	132:12,	% *	00121700
DISPLACEMENT =	145:8,	% THIS VARIABLE IS ADDRESSED	00121800
		% AT THIS DISPLACEMENT	00121900
STRUCTPTRBEFORE=	153:13,	% THIS IS STRUCTURETAB POINTER	00122000
		% BEFORE THIS MESS WAS DECLARED.	00122100
FETCHAUXPTR=	166:13,	% THIS FIELD IS USED FOR	00122200
		% AUXILIARY FETCH DATA IN	00122300

	% ACCESS TYPES AND ACCESS	00122400
	% VARIABLES.	00122500
STOREAUXPTR= 179:13,	% AUXILIARY STORE DATA.	00122600
LABELSEGMENT = 153:15,	% THESE THREE FIELDS	00122700
LABELSTARTADDRESS=166:13,	% DESCRIBE THE BOUNDS OF A	00122800
LABELENDADDRESS=179:13);	% LABEL	00122900
\$TAELE(LEAVESTACK, 100, MARK=1:1, ADDRESS=2:46);	% LEAVESTACK SAVES BRANCH	00123000
	% ADDRESSES FOR A PLANT AT	00123100
	% THE END OF A LABELLED	00123200
	% STATEMENT.	00123300
		00123400
\$ADDCOL(BIGTAB,		00123500
ISDEFINED = 1:1,	% THIS ID IS CURRENTLY	00123600
	% DECLARED. (NOTE: THIS FIELD	00123700
	% ALSO OCCURS IN IDSTACK.)	00123800
ISREINGDEFINED=2:1,	% THIS ID IS DECLARED IN THE	00123900
	% CURRENT BLOCK AND MAY NOT BE	00124000
	% RE-DECLARED BY ANOTHER	00124100
	% DECLARATION IN THE SAME BLOCK	00124200
	% HEAD. (NOTE: THIS FIELD ALSO	00124300
	% OCCURS IN IDSTACK.)	00124400
IDSTACKPTR = 4:12,	% THIS FIELD CONTAINS A	00124500
	% POINTER INTO THE CURRENT ENTRY	00124600
	% IN IDSTACK THAT DEFINES THE	00124700
	% TYPE AND PARAMETERS OF THIS	00124800
	% ID. (NOTE: THIS FIELD ALSO	00124900
	% OCCURS IN IDSTACK.)	00125000
FORWARDDEFINED=3:1,	% THIS IS A FORWARD DEFINED	00125100
	% PROCEDURE IF THIS FIELD IS A	00125200
	% ONE.	00125300
IDLENGTH = 16:6;	% THIS IS THE LENGTH	00125400
	% OF THE ID IN THIS BEGTAB CELL.	00125500
MSTACK,		00125600
OSL2SEMFIELD = 1:35,	% THIS IS THE OSL2	00125700
	% SEMANTICS FIELD IN MSTACK.	00125800
NODE = 1:1,	% IF THIS BIT IS ON, THE ID	00125900
	% SCANNED IS OF CLASS NODE.	00126000
STRUCTUREPTR = 2:13,	% THIS FIELD POINTS INTO	00126100
	% STRUCTURETAR IF THE ID	00126200
	% SCANNED WAS A STRUCTURE.	00126300
NUMBRSUBSTRUCT=15:8,	% LENGTH OF ABOVE VECTOR.	00126400
PROCEDUREPTRISINTOIDSTACK = 23:1,		00126500
	% IF PROCEDUREPTR BELOW	00126600
	% POINTS TO A PROCEDURE IN THE	00126700
	% IDSTACK RATHER THAN	00126800
	% STRUCTURETAR, THEN THIS	00126900
	% FIELD IS SET TO ONE (1).	00127000
PROCEDUREPTR = 24:12);	% POINTER TO PROCEDURE	00127100
	% IF LAST ID SCANNED WAS A	00127200
	% PROCEDURE.	00127300
\$STACK(IDCOUNTERSTACK, 32;	% THIS STACK CONTAINS ID	00127400
	% COUNTERS THAT TELL HOW MANY	00127500
	% IDENTIFIERS HAVE TO BE POPPED	00127600
	% OFF OF THE ID STACK AT THE	00127700
	% CLOSE OF THE CURRENT BLOCK.	00127800
STRUCTUREPARAMSTACK, 160;	% THIS STACK HOLDS PUSHED DOWN	00127900
	% PARAMETERS DURING A STRUCTURE	00128000
	% DECLARATION SCAN.	00128100
PROCEDUREIDSTACK, 32;	% AS NESTED PROCEDURE DECLARA-	00128200
	% TIONS ARE ENCOUNTERED, THE	00128300
	% POINTER TO THE OLD PROCEDURE	00128400

```

% BEING DEFINED IS PUSHED INTO 00128500
% THIS STACK, THE PURPOSE OF 00128600
% THE POINTERS IS TO ALLOW 00128700
% RETURN STATEMENTS TO BE 00128800
% PROPERLY PROCESSED. 00128900
BOOLEAN:FORMALSWITCHSTACK, 32;% THIS STACK IS USED TO STORE 00129000
% THE STATUS OF THE SCANNING= 00129100
% FORMALPARAMETERDECLARATIONS 00129200
% SWITCH 00129300
% CONTAINS PUSHED DOWN VALUES 00129400
% CURRENTNUMBEROFSURS AND 00129500
% OTHER DATA. 00129600
% HOLDS SEGMENT AND 00129700
% ADDRESS DATA 00129800
% HOLDS IF INFORMATION 00129900
% FOR IF STATEMENTS 00130000
% HOLDS IF INFORMATION 00130100
% FOR IF EXPRESSIONS 00130200
% THIS STACK HOLDS THE 00130300
% NUMBER OF LEFT PART ITEMS 00130400
% TO BE STORED INTO. 00130500
% THIS STACK HOLDS FRANCH 00130600
% ADDRESSES FOR CASE OBJECT 00130700
% CODE. 00130800
% WHEN SCANNING 00130900
% ACTUAL PARAMETERS, THIS 00131000
% STACK CONTAINS THE PUSHED 00131100
% DOWN VALUES OF THE FORMAL 00131200
% AND ACTUAL PARAMETER 00131300
% COUNTERS AND POINTERS. 00131400
% IDSTACK ARE STORED HERE. 00131500
00200100
DEFINE PRINTID = 00200200
    POINTER($OPD(BIGTAB,$OPD(MSTACK,$TOP,ENTRY)+1,$ALL))00200300
    FOR $OPD(BIGTAB,$OPD(MSTACK,$TOP,ENTRY),IDLENGTH) #;00200400
        % THIS DEFINE PRINTS THE ID 00200500
        % CHARACTERS POINTED TO IN 00200600
        % BIGTAB BY THE ENTRY FIELD 00200700
        % IN THE TOP OF MSTACK. 00300100
00300200
00300300
00300400
PROCEDURE DUMPENTRY(ID,ENTRY,INDENT); 00300500
VALUE ID,ENTRY,INDENT; 00300600
BOOLEAN ID; INTEGER ENTRY,INDENT; 00300700
BEGIN 00300800
    INTEGER I,NFL,NOL,N,T; 00300900
    ALPHA ARRAY A(0:89); 00301000
    POINTER P,PA; 00301100
    BLANKMFSOUT; 00301200
    PA ← POINTER(A); 00301300
    REPLACE P:PMSOUT+(I+MIN(5×INDENT,95)) BY "===> "; 00301400
    IF ID THEN REPLACE P:P BY "IDSTACK " 00301500
        ELSE REPLACE P:P BY "STRUCTURETAB "; 00301600
    REPLACE P BY "ENTRY ",ENTRY FOR 4 DIGITS, 00301700
        " LEVEL ", INDENT FOR 2 DIGITS; 00301800
    WRITEMFSOUT; 00301900
    I ← I+5; 00302000
    NFL ← ENTIER((132-I)/20); 00302100
    P ← PA; 00302200
    IF (T ← $OPD(IDENTRY,TYPE)) ≠ 0 00302300
    THEN

```

```

BEGIN
    IF T > 13
    THEN REPLACE P:P BY "TYPE.....",T FOR 2 DIGITS
    ELSE
    BEGIN
        DEFINE RPP=REPLACE P:P BY #;
        RPP "TYPE.....";
        CASE T OF
        BEGIN
            REPLACE P:P-2 BY "UNDECLARED";
            RPP ".INTEGER";
            RPP ".BOOLEAN";
            RPP "...STRING";
            RPP ".POINTER";
            RPP "...EVENT";
            RPP ".PROCESS";
            RPP ".PATTERN";
            REPLACE P:P-1 BY "STRUCTURE";
            RPP "...ACCESS";
            RPP "....CODE";
            RPP "....FILE";
            RPP "...LABEL";
            REPLACE P:P-1 BY "ACCESSVAR";
        END
    END
END;

IF (T + $OPD(IDENTRY,ACCESSTYPE)) # 0
THEN REPLACE P:P BY "ACCESSTYPE.....",T FOR 2 DIGITS;
IF (T + $OPD(IDENTRY,STORETYPE)) # 0
THEN REPLACE P:P BY "STORETYPE.....",T FOR 2 DIGITS;
IF (T + $OPD(IDENTRY,FETCHTYPE)) # 0
THEN REPLACE P:P BY "FETCHTYPE.....",T FOR 2 DIGITS;
IF (T + $OPD(IDENTRY,NUMBROFSUPS)) # 0
THEN REPLACE P:P BY "NUMBROFSUPS....",T FOR 3 DIGITS;
IF (T + $OPD(IDENTRY,NUMBRSUBSTRUCT)) # 0
THEN REPLACE P:P BY "NUMBRSUBSTRUCT.",T FOR 3 DIGITS;
IF (T + $OPD(IDENTRY,NUMBROFPARMS)) # 0
THEN REPLACE P:P BY "NUMBROFPARMS...",T FOR 3 DIGITS;
IF BOOLEAN($OPD(IDENTRY,MACRO))
THEN REPLACE P:P BY "MACRO.....TRUE";
IF BOOLEAN($OPD(IDENTRY,SUBSCRIPTED))
THEN REPLACE P:P BY "SUBSCRIPTED...TRUE";
IF BOOLEAN($OPD(IDENTRY,PROCEDURE))
THEN REPLACE P:P BY "PROCEDURE.....TRUE";
IF BOOLEAN($OPD(IDENTRY,NODE))
THEN REPLACE P:P BY "NODE.....TRUE";
IF BOOLEAN($OPD(IDENTRY,FORMALPARM))
THEN REPLACE P:P BY "FORMALPARM....TRUE";
IF BOOLEAN($OPD(IDENTRY,VALUEPARM))
THEN REPLACE P:P BY "VALUEPARM.....TRUE";
IF BOOLEAN($OPD(IDENTRY,HASRENSPEC))
THEN REPLACE P:P BY "HASRENSPEC...TRUE";
IF BOOLEAN($OPD(IDENTRY,OWN))
THEN REPLACE P:P BY "OWN.....TRUE";
IF (T + $OPD(IDENTRY,BIGTARPTR)) # 0
THEN REPLACE P:P BY "BIGTARPTR.....",T FOR 4 DIGITS;
IF (T + $OPD(IDENTRY,STRUCTUREPTR)) # 0
THEN REPLACE P:P BY "STRUCTUREPTR...",T FOR 4 DIGITS;
IF (T + $OPD(IDENTRY,FORMALPARMPTR)) # 0
THEN REPLACE P:P BY "FORMALPARMPTR.",T FOR 4 DIGITS;
IF (T + $OPD(IDENTRY,FETCHPROCEDURE)) # 0

```

```

THEN REPLACE P:P BY "FFETCHPROCEDURE..", T FOR 4 DIGITS;
IF (T < $OPD(IDENTRY, STOREPROCEDURE)) ≠ 0
THEN REPLACE P:P BY "STOPPROCEDURE..", T FOR 4 DIGITS;
IF (T < $OPD(IDENTRY, FETCHAUXPTR)) ≠ 0
THEN REPLACE P:P BY "FFETCHAUXPTR...", T FOR 4 DIGITS;
IF (T < $OPD(IDENTRY, STOREAUXPTR)) ≠ 0
THEN REPLACE P:P BY "STOREAUXPTR...", T FOR 4 DIGITS;
IF BOOLEAN($OPD(IDENTRY, ISDEFINED))
THEN REPLACE P:P BY "ISDEFINED.....TRUE";
IF BOOLEAN($OPD(IDENTRY, ISREINGDEFINED))
THEN REPLACE P:P BY "ISREINGDEF.....TRUE";
IF BOOLEAN($OPD(IDENTRY, FORWARDDEFINED))
THEN REPLACE P:P BY "FORWARDDEF.....TRUE";
IF (T < $OPD(IDENTRY, IDSTACKPTR)) ≠ 0
THEN REPLACE P:P BY "IDSTACKPTR...", T FOR 4 DIGITS;
IF (T < $OPD(IDENTRY, STRUCTPTRPREFOR)) ≠ 0
THEN REPLACE P:P BY "STRUCTPTRPREFOR..", T FOR 4 DIGITS;
N ← DELTA(PA, P)/18;
IF N = 0
THEN
BEGIN
    BLANKMESOUT;
    REPLACE PMESOUT+I BY "NO ENTRIES";
    WRITEMESOUT
END
ELSE
BEGIN
    T ← 0;
    WHILE T < N DO
    BEGIN
        BLANKMESOUT;
        P ← PMESOUT+I;
        NOL ← 0;
        WHILE NOL < NPL AND T < N DO
        BEGIN
            REPLACE P:P BY PA+18×T FOR 18, " ";
            NOL ← NOL+1;
            T ← T+1;
        END;
        WRITEMESOUT
    END
END
END DUMPENTRY;

PROCEDURE DUMPBIGTAB(N, INDENT); VALUE N, INDENT; INTEGER N, INDENT;
BEGIN
    INTEGER I, J;
    BLANKMESOUT;
    REPLACE PMESOUT+(J+MIN(5×INDENT, 100)) BY "====> BIGTAB ",
        N FOR 4 DIGITS,
        " ID=", POINTER($OPD(BIGTAB, N+1, $ALL))
        FOR MIN($OPD(BIGTAB, N, IDLENGTH), 112-I);
    WRITEMESOUT;
    I ← I+5;
    IF BOOLEAN($OPD(BIGTAB, N, ISDEFINED))
    THEN
    BEGIN
        BLANKMESOUT;
        REPLACE PMESOUT+I BY "ISDEFINED";
    
```

00308500
 00308600
 00308700
 00308800
 00308900
 00309000
 00309100
 00309200
 00309300
 00309400
 00309500
 00309600
 00309700
 00309800
 00309900
 00310000
 00310100
 00310200
 00310300
 00310400
 00310500
 00310600
 00310700
 00310800
 00310900
 00311000
 00311100
 00311200
 00311300
 00311400
 00311500
 00311600
 00311700
 00311800
 00311900
 00312000
 00312100
 00312200
 00312300
 00312400
 00312500
 00312600
 00312700
 00312800
 00312900
 00313000
 00313100
 00313200
 00313300
 00313400
 00313500
 00313600
 00313700
 00313800
 00313900
 00314000
 00314100
 00314200
 00314300
 00314400
 00314500

```

WRITEMESOUT                                00314600
END;                                         00314700
IF BOOLEAN($OPD(BIGTAB,N,ISREINGDEFINED))  00314800
THEN                                         00314900
BEGIN                                       00315000
    BLANKMESOUT;                           00315100
    REPLACE PMESOUT+I BY "ISBFINGODEFINED"; 00315200
    WRITEMESOUT                             00315300
END;                                         00315400
IF BOOLEAN($OPD(BIGTAB,N,FORWARDDEFINED))  00315500
THEN                                         00315600
BEGIN                                       00315700
    BLANKMESOUT;                           00315800
    REPLACE PMESOUT+I BY "FORWARDDEFINED";  00315900
    WRITEMESOUT                             00316000
END;                                         00316100
IF (J ← $OPD(BIGTAB,N,IDSTACKPTR)) ≠ 0    00316200
THEN                                         00316300
BEGIN                                       00316400
    BLANKMESOUT;                           00316500
    REPLACE PMESOUT+I BY "IDSTACKPTR=", J FOR 4 DIGITS; 00316600
    WRITEMESOUT                             00316700
END;                                         00316800
END DUMPBIGTAB;                            00316900
                                           00317000
                                           00317100
                                           00317200
PROCEDURE DUMPTAB(ID,ENTRY,TRACEPARM,TRACESEM,INDENT,TRACEBIG); 00317300
VALUE JD,ENTRY,TRACEPARM,TRACESEM,INDENT,TRACEBIG; 00317400
BOOLEAN ID, % TRUE IF IDSTACK = OTHERWISE STRUCTURETAB 00317500
TRACEPARM, % TRUE IF FORMALS AND SUBSTRUCT TO BE DUMPED 00317600
TRACESEM, % TRUE IF OLDSEMFIELD TO BE TRACED 00317700
TRACEBIG; % TRUE IF BIGTAB TO BE DUMPED 00317800
INTEGER ENTRY, % WHICH ENTRY TO DUMP 00317900
INDENT; % LEVEL OF RECURSION 00318000
BEGIN                                       00318100
    INTEGER I,J,NS,SP,NP,PP,IP,BT; 00318200
    LABEL AWAY; 00318300
    IF ENTRY = 0 THEN GO AWAY; 00318400
    IF ID THEN $ENTER(IDENTRY,$ALLOF(IDSTACK,ENTRY)) 00318500
    ELSE $ENTER(IDENTRY,$ALLOF(STRUCTURETAB,ENTRY)); 00318600
    NS ← $OPD(IDENTRY,NUMBRSUPSTRUCT); 00318700
    SP ← $OPD(IDENTRY,STRUCTUREPTR); 00318800
    NP ← $OPD(IDENTRY,NUMBROFPARMS); 00318900
    PP ← $OPD(IDENTRY,FORMALPARMPTR); 00319000
    IP ← $OPD(IDENTRY,IDSTACKPTR); 00319100
    J ← MIN(5×INDENT,95)+5; 00319200
    BT ← $OPD(IDENTRY,BIGTABPTR); 00319300
    DUMFENTRY(ID,ENTRY,INDENT); 00319400
    IF TRACEBIG THEN DUMPBIGTAB(BT,INDENT+1); 00319500
    IF TRACEPARM 00319600
    THEN 00319700
    BEGIN 00319800
        IF NS ≠ 0 00319900
        THEN 00320000
        BEGIN 00320100
            BLANKMESOUT; 00320200
            REPLACE PMESOUT+J BY 00320300
                "***** SUBSTRUCTURES *****"; 00320400
            WRITEMESOUT; 00320500
            FOR I ← 1 STEP 1 UNTIL NS DO 00320600

```

```

        DUMPTAB(FALSE,SP+1-I,
                TRACEPARM,TRACESEM,INDENT+1,TRACEBIG)
    END;
    IF NP ≠ 0
    THEN
    BEGIN
        BLANKMESOUT;
        REPLACE PMESOUT+J BY
            "***** FORMAL PARAMETERS *****";
        WRITEMESOUT;
        FOR I + 1 STEP 1 UNTIL NP DO
            DUMPTAB(FALSE,PP+1-I,
                    TRACEPARM,TRACESEM,INDENT+1,TRACEBIG)
        END;
    END TRACEPARM;
    IF TRACESEM AND IP≠0
    THEN
    BEGIN
        BLANKMESOUT;
        REPLACE PMESOUT+J BY
            "***** UP LEVEL ID *****";
        WRITEMESOUT;
        DUMPTAB(TRUE,IP,TRACEPARM,TRACESEM,INDENT+1,TRACEBIG)
    END TRACESEM;
    AWAY;
    END DUMPTAB;

PROCEDURE DUMPMSTACK(N); VALUE N; INTEGER N;
BEGIN
    POINTER P;
    INTEGER I;
    DEFINE COMMA=IF P≠PMESOUT+5 THEN REPLACE P:P BY ", ";
    BLANKMESOUT;
    REPLACE PMESOUT BY "====> MSTACK ", N FOR 4 DIGITS;
    WRITEMESOUT;
    BLANKMESOUT;
    P + PMESOUT+5;
    IF BOOLEAN($OPD(MSTACK,N,NODE))
    THEN REPLACE P:P BY "NODE";
    IF BOOLEAN($OPD(MSTACK,N,PROCEDUREPTRISINTOIDSTACK))
    THEN
    BEGIN
        COMMA;
        REPLACE P:P BY "PROCEDUREPTRISINTOIDSTACK"
    END;
    IF (I + $OPD(MSTACK,N,PROCEDUREPTR)) ≠ 0
    THEN
    BEGIN
        COMMA;
        REPLACE P:P BY "PROCEDUREPTR=", I FOR 4 DIGITS
    END;
    IF (I + $OPD(MSTACK,N,ENTRY)) ≠ 0
    THEN
    BEGIN
        COMMA;
        REPLACE P:P BY "ENTRY=", I FOR 4 DIGITS;
    END;
    IF (I + $OPD(MSTACK,N,STRUCTUREPTR)) ≠ 0
    THEN

```

```

00320700
00320800
00320900
00321000
00321100
00321200
00321300
00321400
00321500
00321600
00321700
00321800
00321900
00322000
00322100
00322200
00322300
00322400
00322500
00322600
00322700
00322800
00322900
00323000
00323100
00323200
00323300
00323400
00323500
00323600
00323700
00323800
00323900
00324000
00324100
00324200
00324300
00324400
00324500
00324600
00324700
00324800
00324900
00325000
00325100
00325200
00325300
00325400
00325500
00325600
00325700
00325800
00325900
00326000
00326100
00326200
00326300
00326400
00326500
00326600
00326700

```

```

BEGIN                                                    00326800
    COMMA;                                              00326900
    REPLACE P:P BY "STRUCTUREPTR=", I FOR 4 DIGITS;    00327000
END;                                                    00327100
IF (I + $OPD(MSTACK,N,NUMBRSUBSTRUCT)) ≠ 0            00327200
THEN                                                    00327300
BEGIN                                                  00327400
    COMMA;                                              00327500
    REPLACE P:P BY "NUMBRSUBSTRUCT=", I FOR 3 DIGITS  00327600
END;                                                    00327700
WRITEMESOUT;                                           00327800
END DUMPMSTACK;                                        00327900
                                                    00328000
                                                    00328100
                                                    00328200
PROCEDURE PRINTSTACK(A,B); VALUE B; ALPHA ARRAY A(0); INTEGER R; 00328300
BEGIN                                                    00328400
    INTEGER I;                                          00328500
    FORMAT F(X10,I10);                                00328600
    FOR I ← R-1 STEP -1 UNTIL 0 DO                     00328700
        WRITE(LINE,F,A[I]);                            00328800
END;                                                    00328900
                                                    00329000
                                                    00329100
                                                    00329200
PROCEDURE PRINTBSTACK(A,R);VALUE B;BOOLEAN ARRAY A(0);INTEGER B; 00329300
BEGIN                                                    00329400
    INTEGER I;                                          00329500
    FORMAT F(X15,L5);                                  00329600
    FOR I ← R-1 STEP -1 UNTIL 0 DO                     00329700
        WRITE(LINE,F,A[I]);                            00329800
END;                                                    00329900
                                                    00330000
                                                    00330100
                                                    00330200
BOOLEAN PROCEDURE LANGUAGECONTROLCARD(PARM);           00330300
VALUE PARM; ALPHA PARM;                                00330400
COMMENT THIS PROCEDURE IS TRUE IF AN DSL2 CONTROL CARD IS 00330500
BEING SCANNED. OTHERWISE IT IS FALSE. THE LEGITIMATE DSL2 CONTROL 00330600
CARDS ARE AS FOLLOWS:
    #DUMP [IDSTACK/STRUCTURETAB] [TRACE([PARM/SEM/PIGTAB],&)]& 00330800
        [ALL/N/TOP N]                                00330900
        IDSTACK OR STRUCTURETAB IS DUMPED.            00331000
        IF TRACE IS NOT PRESENT, THEN ONLY THE        00331100
        INDICATED ENTRIES ARE DUMPED.                 00331200
        IF TRACE IS ON, THEN THE TRACE PARAMETERS     00331300
        CAUSE A TRACE ON THE INDICATED POINTERS       00331400
        PARM => STRUCTUREPTR & FORMALPARMPTR          00331500
        SEM => OLD SEMFIELD                           00331600
        BIGTAB => RIGTABPTR                            00331700
        IF NO TRACE PARAMETER IS ENTERED, THEN ALL    00331800
        ARE ASSUMED.                                  00331900
        ONE CAN DUMP ALL OF THE ENTRIES IN A          00332000
        STACK OR JUST ENTRY N OR THE TOP N ENTRIES.   00332100
MSTACK [TOP N/TRACE [RIGTAB/PARM/SEM/ALL]]            00332200
    THE TOP N ELEMENTS OF MSTACK ARE DUMPED           00332300
    OR THE TOP ELEMENT IS DUMPED WITH A TRACE         00332400
    INTO JUST RIGTAB OR IF PARM OR SEM OR ALL         00332500
    ARE SPECIFIED, THEN THE TRACE CONTINUES INTO      00332600
    IDSTACK OR STRUCTURETAB WITH THE INDICATED        00332700
    TRACE PARAMETERS. NOTE THAT THE USE OF            00332800

```

BIGTAB HERE IS UNIQUE. THE ONLY WAY TO TURN
ON A BIGTAB TRACE PARAMETER WHEN A TRACE
PROCEEDS INTO IDSTACK OR STRUCTURETAB IS TO
USE THE TRACE PARAMETER ALL WHICH TURNS ON
ALL OF THE TRACE PARAMETERS.

IDCOUNTERSTACK
STRUCTUREPARMSTACK
PROCEDUREIDSTACK
SUBSCRIPTSTACK
ACTUALPARMSTACK
FORMALSWITCHSTACK
STATUS

THIS DUMPS ALL OF THE
SINGLE VARIABLES IF THEY
ARE NON ZERO OR TRUE

@OBJECT TURNS ON OBJECT PRINTING
@NOOBJECT TURNS OFF OBJECT PRINTING
@KILL DOES A DIVIDE BY ZERO;

BEGIN

BOOLEAN ID,TRACEPARM,TRACESEM,TRACEBIG;
INTEGER N,TOP,BOTTOM;

LANGUAGECONTROLCARD + TRUE;

IF PARM = "KILL "

THEN I + I/O

ELSE IF PARM = "OBJECT" THEN PRINTOBJECT + TRUE

ELSE IF PARM = "NOOBJECT" THEN PRINTOBJECT + FALSE

ELSE IF PARM="DUMP "

THEN

DO BEGIN

DEFINE PRINT = BLANKMESOUT;REPLACE PMESOUT+5 BY #,

FINISH(FINISH1)=," = ",

FINISH1 FOR 4 DIGITS; WRITEMESOUT#,

PRINTLABEL=BLANKMESOUT;

REPLACE PMESOUT BY "===> ",#;

LABEL FANCY;

LANGUAGECONTROLCARD + TRUE;

PARM + CCSCAN;

IF PARM = "STATUS"

THEN

BEGIN

PRINTLABEL "STATUS";

WRITEMESOUT;

IF IDCOUNTER # 0

THEN

BEGIN

PRINT "IDCOUNTER"

FINISH(IDCOUNTER)

END;

IF IDLISTCOUNTER # 0

THEN

BEGIN

PRINT "IDLISTCOUNTER"

FINISH(IDLISTCOUNTER)

END;

IF CURRENTDECLTYPE # 0

THEN

BEGIN

PRINT "CURRENTDECLTYPE"

FINISH(CURRENTDECLTYPE)

END;

IF CURRENTDECLDOWN # 0

THEN

00332900

00333000

00333100

00333200

00333300

00333400

00333500

00333600

00333700

00333800

00333900

00334000

00334100

00334200

00334300

00334400

00334500

00334600

00334700

00334800

00334900

00335000

00335100

00335200

00335300

00335400

00335500

00335600

00335700

00335800

00335900

00336000

00336100

00336200

00336300

00336400

00336500

00336600

00336700

00336800

00336900

00337000

00337100

00337200

00337300

00337400

00337500

00337600

00337700

00337800

00337900

00338000

00338100

00338200

00338300

00338400

00338500

00338600

00338700

00338800

00338900

BEGIN	00339000
PRINT "CURRENTDECLOWN"	00339100
FINISH(CURRENTDFCLOWN)	00339200
END;	00339300
IF CURRENTDECLNODE # 0	00339400
THEN	00339500
BEGIN	00339600
PRINT "CURRENTDFCLNODE"	00339700
FINISH(CURRENTDFCLNODE)	00339800
END;	00339900
IF CURRENTDECLMACRO # 0	00340000
THEN	00340100
BEGIN	00340200
PRINT "CURRENTDFCLMACRO"	00340300
FINISH(CURRENTDFCLMACRO)	00340400
END;	00340500
IF CURRENTNUMPROFSUBS # 0	00340600
THEN	00340700
BEGIN	00340800
PRINT "CURRENTNUMPROFSUBS"	00340900
FINISH(CURRENTNUMPROFSUBS)	00341000
END;	00341100
IF CURRENTPROCEDUREID # 0	00341200
THEN	00341300
BEGIN	00341400
PRINT "CURRENTPROCEDUREID"	00341500
FINISH(CURRENTPROCEDUREID)	00341600
END;	00341700
IF CURRENTEXPRESSIONTYPE # 0	00341800
THEN	00341900
BEGIN	00342000
PRINT "CURRENTEXPRESSIONTYPE"	00342100
FINISH(CURRENTEXPRESSIONTYPE)	00342200
END;	00342300
IF CURRENTSTRUCTUREPTR # 0	00342400
THEN	00342500
BEGIN	00342600
PRINT "CURRENTSTRUCTUREPTR"	00342700
FINISH(CURRENTSTRUCTUREPTR)	00342800
END;	00342900
IF CURRENTNUMBEROFSUBSTRUCTURES # 0	00343000
THEN	00343100
BEGIN	00343200
PRINT "CURRENTNUMBEROFSUBSTRUCTURES"	00343300
FINISH(CURRENTNUMBEROFSUBSTRUCTURES)	00343400
END;	00343500
IF CURRENTIDPTR # 0	00343600
THEN	00343700
BEGIN	00343800
PRINT "CURRENTIDPTR"	00343900
FINISH(CURRENTIDPTR)	00344000
END;	00344100
IF CURRENTFORMALPARAMETERLISTPOINTER # 0	00344200
THEN	00344300
BEGIN	00344400
PRINT "CURRENTFORMALPARAMETERLISTPOINTER"	00344500
FINISH(CURRENTFORMALPARAMETERLISTPOINTER)	00344600
END;	00344700
IF CURRENTNUMBEROFFORMALPARAMETERS # 0	00344800
THEN	00344900
BEGIN	00345000

PRINT "CURRENTNUMBEROFFORMALPARAMETERS"	00345100
FINISH(CURRENTNUMBEROFFORMALPARAMETERS)	00345200
END;	00345300
IF CURRENTACTUALPARAMETERCOUNT # 0	00345400
THEN	00345500
BEGIN	00345600
PRINT "CURRENTACTUALPARAMETERCOUNT"	00345700
FINISH(CURRENTACTUALPARAMETERCOUNT)	00345800
END;	00345900
IF CURRENTMAXACCESSNUMBER # 0	00346000
THEN	00346100
BEGIN	00346200
PRINT "CURRENTMAXACCESSNUMBER"	00346300
FINISH(CURRENTMAXACCESSNUMBER)	00346400
END;	00346500
IF CURRENTACCESSID # 0	00346600
THEN	00346700
BEGIN	00346800
PRINT "CURRENTACCESSID"	00346900
FINISH(CURRENTACCESSID)	00347000
END;	00347100
IF STRUCTUREISOWN	00347200
THEN	00347300
BEGIN	00347400
PRINT"STRUCTUREISOWN";	00347500
WRITEMESOUT	00347600
END;	00347700
IF SCANNINGFORMALPARAMETERDECLARATIONS	00347800
THEN	00347900
BEGIN	00348000
PRINT"SCANNINGFORMALPARAMETERDECLARATIONS";	00348100
WRITEMESOUT	00348200
END;	00348300
IF CURRENTSTRUCTUREPTRISINIDSTACK	00348400
THEN	00348500
BEGIN	00348600
PRINT"CURRENTSTRUCTUREPTRISINIDSTACK";	00348700
WRITEMESOUT	00348800
END;	00348900
END	00349000
ELSE IF PARM = "IDCOUN"	00349100
THEN	00349200
BEGIN	00349300
PRINTLABEL "IDCOUNTERSTACK";	00349400
WRITEMESOUT;	00349500
PRINTSTACK(IDCOUNTERSTACK,PTIDCOUNTERSTACK)	00349600
END	00349700
ELSE IF PARM = "STRUKT"	00349800
THEN	00349900
BEGIN	00350000
PRINTLABEL "STRUCTUREPARMSTACK";	00350100
WRITEMESOUT;	00350200
PRINTSTACK(STRUCTUREPARMSTACK,PTSTRUCTUREPARMSTACK)	00350300
END	00350400
ELSE IF PARM = "PROCED"	00350500
THEN	00350600
BEGIN	00350700
PRINTLABEL "PROCEDUREIDSTACK";	00350800
WRITEMESOUT;	00350900
PRINTSTACK(PROCEDUREIDSTACK,PTPROCEDUREIDSTACK)	00351000
END	00351100

```

ELSE IF PARM = "SUBSCR"      00351200
THEN                          00351300
BEGIN                          00351400
    PRINTLAPL "SUBSCRIPTSTACK"; 00351500
    WRITEMESOUT;                00351600
    PRINTSTACK(SUBSCRIPTSTACK,PTSUBSCRIPTSTACK) 00351700
END                            00351800
ELSE IF PARM = "ACTUAL"      00351900
THEN                          00352000
BEGIN                          00352100
    PRINTLAPL "ACTUALPARMSTACK"; 00352200
    WRITEMESOUT;                00352300
    PRINTSTACK(ACTUALPARMSTACK,PTACTUALPARMSTACK) 00352400
END                            00352500
ELSE IF PARM = "FORMAL"      00352600
THEN                          00352700
BEGIN                          00352800
    PRINTLAPL "FORMALSWITCHSTACK"; 00352900
    WRITEMESOUT;                00353000
    PRINTSTACK(FORMALSWITCHSTACK,PTFORMALSWITCHSTACK) 00353100
END                            00353200
ELSE IF PARM = "IDSTAC"      00353300
THEN                          00353400
BEGIN                          00353500
    ID ← TRUE;                  00353600
    GO TO FANCY                 00353700
END                            00353800
ELSE IF PARM = "STRUCT"      00353900
THEN                          00354000
BEGIN                          00354100
    ID ← FALSE;                00354200
FANCY: PARM ← CCSCAN;          00354300
    IF PARM = "TRACE "         00354400
    THEN                       00354500
    DO BEGIN                   00354600
        PARM ← CCSCAN;        00354700
        IF PARM = "PARM "     00354800
        THEN                  00354900
        BEGIN                 00355000
            TRACEPARM ← TRUE;  00355100
            PARM ← CCSCAN      00355200
        END                   00355300
        ELSE IF PARM = "SEM "  00355400
        THEN                  00355500
        BEGIN                 00355600
            TRACESEM ← TRUE;   00355700
            PARM ← CCSCAN      00355800
        END                   00355900
        ELSE IF PARM = "FIGTAB" 00356000
        THEN                  00356100
        BEGIN                 00356200
            TRACERIG ← TRUE;   00356300
            PARM ← CCSCAN      00356400
        END                   00356500
        ELSE TRACEPARM ← TRACESEM ← TRACERIG ← TRUE 00356600
    END UNTIL PARM ≠ " ";      00356700
    IF PARM = "ALL "          00356800
    THEN                       00356900
    BEGIN                      00357000
        TOP ← IF ID THEN PTIDSTACK-1 00357100
            ELSE PTSTRUCTUREFAR-1; 00357200
    END

```

```

        BOTTOM + 1;
END
ELSE IF PARM = "TOP"
THEN
BEGIN
    TOP + IF ID THEN PTIDSTACK-1
           ELSE PTSTRUCTURETAB-1;
    N + CCSCAN;
    BOTTOM + TOP+1-N;
END
ELSE TOP + BOTTOM + PARM;
FOR N + TOP STEP -1 UNTIL BOTTOM DO
DUMPTAB(ID,N,TRACEPARM,TRACESEM,0,TRACEBIG)
END
ELSE IF PARM = "MSTACK"
THEN
BEGIN
    IF (PARM + CCSCAN) = "TRACE"
    THEN
    BEGIN
        DO BEGIN
            IF (PARM + CCSCAN) = "ALL"
            THEN TRACEPARM+TRACESEM+TRACEBIG+TRUE
            ELSE IF PARM = "PARM"
            THEN TRACEPARM + TRUE
            ELSE IF PARM = "SEM"
            THEN TRACESEM + TRUE;
        END UNTIL (PARM + CCSCAN) ≠ " ";
        DUMPMSTACK(PTMSTACK-1);
        IF TRACEPARM OR TRACESEM OR TRACEBIG
        THEN DUMPTAB(TRUE,$OPD(BIGTR,
                                $OPD(MSTACK,$TOP,ENTRY),
                                IDSTACKPTR),
                                TRACEPARM,TRACESEM,1,TRACEBIG);
    END
    ELSE IF PARM = "TOP"
    THEN
    BEGIN
        N + CCSCAN;
        BOTTOM + MAX(PTMSTACK-N,0);
        TOP + PTMSTACK-1;
        FOR N + TOP STEP -1 UNTIL BOTTOM DO
        DUMPMSTACK(N);
    END
END
ELSE
BEGIN
    BLANKMESOUT;
    REPLACE PMESOUT BY "==> I CAN NOT UNDERSTAND ",
        "THIS $DUMP CARD.";
    WRITEMESOUT
END;
IF CCSCAN = ";" THEN ID+LANGUAGECONTROLCARD("DUMP ");
END UNTIL CCSCAN ≠ ";";
ELSE LANGUAGECONTROLCARD + FALSE
END LANGUAGECONTROLCARD;

PROCEDURE NOFORWARDMATCH;
BEGIN

```

```

00357300
00357400
00357500
00357600
00357700
00357800
00357900
00358000
00358100
00358200
00358300
00358400
00358500
00358600
00358700
00358800
00358900
00359000
00359100
00359200
00359300
00359400
00359500
00359600
00359700
00359800
00359900
00360000
00360100
00360200
00360300
00360400
00360500
00360600
00360700
00360800
00360900
00361000
00361100
00361200
00361300
00361400
00361500
00361600
00361700
00361800
00361900
00362000
00362100
00362200
00362300
00362400
00362500
00362600
00362700
00362800
00400100
00400200
00400300
00400400
00400500

```

```

ERRMS "THIS PROCEDURE HEADING DOES NOT MATCH THE FORWARD ", 00400600
      "DECLARATION OF ITSELF."); 00400700
WRITEMESOUT; 00400800
END NOFORWARDMATCH; 00400900
                                00401000
                                00401100
                                00401200
                                00401300
                                00401400
                                00401500
                                00401600
                                00401700
                                00401800
                                00401900
                                00402000
                                00402100
                                00402200
                                00402300
                                00402400
                                00402500
                                00402600
                                00402700
                                00402800
                                00402900
                                00403000
                                00403100
                                00403200
                                00403300
                                00403400
                                00403500
                                00403600
                                00403700
                                00403800
                                00403900
                                00404000
                                00404100
                                00404200
                                00404300
                                00404400
                                00404500
                                00404600
                                00404700
                                00404800
                                00404900
                                00405000
                                00405100
                                00405200
                                00405300
                                00405400
                                00405500
                                00405600
                                00405700
                                00405800
                                00405900
                                00406000
                                00406100
                                00406200
                                00406300
                                00406400
                                00406500
                                00406600

BOOLEAN PROCEDURE EQUALSTRUCTURETAB(A,B); INTEGER A,B;
  COMMENT THIS PROCEDURE CHECKS THAT THE DATA TYPES AT A & B IN
  STRUCTURETAB HAVE THE SAME SPECIFICATION AND SUBSTRUCTURE
  SPECIFICATION;
  IF A=0 THEN EQUALSTRUCTURETAB ← B=0
  ELSE IF B=0 THEN EQUALSTRUCTURETAB ← FALSE
  ELSE EQUALSTRUCTURETAB ←
    $OPD(STRUCTURETAB,A,$ALL 0) = $OPD(STRUCTURETAB,B,$ALL 0)
    AND EQUALSTRUCTURETAB($OPD(STRUCTURETAB,A,STRUCTUREPTR),
      $OPD(STRUCTURETAB,B,STRUCTUREPTR))
    AND EQUALSTRUCTURETAB($OPD(STRUCTURETAB,A,FORMALPARMPTR),
      $OPD(STRUCTURETAB,B,FORMALPARMPTR));

BOOLEAN PROCEDURE EQUALIDSTACK(A,B); INTEGER A,B;
  COMMENT THIS PROCEDURE CHECKS IF THE GOODIES AT A & B IN
  IDSTACK HAVE THE SAME SPECIFICATION AND SUBSPECIFICATION INSOFAR
  AS DATA TYPE IS CONCERNED;
  EQUALIDSTACK ← $OPD(IDSTACK,A,$ALL 0)=$OPD(IDSTACK,B,$ALL 0)
    AND EQUALSTRUCTURETAB($OPD(IDSTACK,A,STRUCTUREPTR),
      $OPD(IDSTACK,B,STRUCTUREPTR))
    AND EQUALSTRUCTURETAB($OPD(IDSTACK,A,FORMALPARMPTR),
      $OPD(IDSTACK,B,FORMALPARMPTR));

BOOLEAN PROCEDURE EQUALIDSTRUCT(A,B); INTEGER A,B;
  COMMENT THIS PROCEDURE CHECKS THAT THE GUY A IN IDSTACK IS THE
  SAME TYPE AS B IN STRUCTURETAB;
  EQUALIDSTRUCT ← $OPD(IDSTACK,A,$ALL 0)=$OPD(STRUCTURETAB,B,$ALL 0)
    AND EQUALSTRUCTURETAB($OPD(IDSTACK,A,STRUCTUREPTR),
      $OPD(STRUCTURETAB,B,STRUCTUREPTR))
    AND EQUALSTRUCTURETAB($OPD(IDSTACK,A,FORMALPARMPTR),
      $OPD(STRUCTURETAB,B,FORMALPARMPTR));

BOOLEAN PROCEDURE TESTTYPE(TYPE); VALUE TYPE; INTEGER TYPE;
  COMMENT THIS PROCEDURE CHECKS IF THE TYPE OF THE ID IN MSTACK
  IS THE SAME AS THE "TYPE" IN THE FORMAL PARAMETER LIST;
  BEGIN LABEL FRUDECRP;
    DEFINE RETURN(RETURN1) =
      BEGIN
        TESTTYPE ← RETURN1;
        GO TO FRUDECRP
      END #;
    IF CURRENTSTRUCTUREPTR = 0
    THEN
      BEGIN
        $ENTER(MSTACK,$TOP,NODE;$OPD(IDSTACK,$OPD(FIGTAB,
          $OPD(MSTACK,$TOP,ENTRY),
          IDSTACKPTR) ,NODE));
        IF ( TESTTYPE ←

```

```

$OPD(IDSTACK,$OPD(BIGTAB,$OPD(MSTACK,$TOP,ENTRY),
    IDSTACKPTR),TYPE) = TYPE )
AND (TYPE = STRUCT OR
    $OPD(IDSTACK,
        $OPD(RIGTAB,$OPD(MSTACK,$TOP,ENTRY),
            IDSTACKPTR),NUMPROFSUBS) > 0 )
THEN COUPLE("NAME-CALL",
    $OPD(IDSTACK,I+$OPD(RIGTAB,$OPD(MSTACK,
        $TOP,ENTRY),IDSTACKPTR),LEVEL),
    $OPD(IDSTACK,I,DISPLACEMENT),
    PRINTID);
END
ELSE
BEGIN
    LABEL LOOP;
    INTEGER I;
LOOP:
    BEGIN
        FOR I + 1 STEP 1 UNTIL CURRENTNUMBEROFSTRUCTURES
        DO BEGIN
            IF $OPD(STRUCTURETAB,
                CURRENTSTRUCTUREPTR+1-I,
                BIGTABPTR) =
                $OPD(MSTACK,$TOP,ENTRY)
            THEN LEAVE LOOP
            END;
            ERRMS "THIS ID IS NOT PART OF THIS STRUCTURE.";
            WRITEMESOUT;
            RETURN(TRUE);
        END; LOOP;
        CURRENTIDPTR + CURRENTSTRUCTUREPTR+1-I;
        IF TESTTYPE + $OPD(STRUCTURETAB,CURRENTIDPTR,TYPE)=TYPE
        THEN
            BEGIN
                OPERFIRSTPART,
                "INTEGER-LITERAL-CALL ",
                I-1 FOR 3 DIGITS;
                OPERLASTPART;
                OPERFIRSTPART,
                "MAKE-INDEX ",
                PRINTID;
                OPERLASTPART;
            END;
        END;
    END;
    ; ERUDECORP: END ;

INTEGER PROCEDURE FINDSUBSTRUCTUREID(BIGTABPTR);
VALUE BIGTABPTR; INTEGER RIGTABPTR;
COMMENT THIS PROCEDURE USES THE CURRENT VALUES OF
CURRENTSTRUCTUREPTR AND CURRENTNUMBEROFSTRUCTURES TO FIND
AN ENTRY IN STRUCTURETAB THAT HAS THE SAME ID AS THAT WHICH
IS AT RIGTABPTR IN RIGTAB. IF NO ENTRY IS FOUND, THEN THE
VALUE OF FINDSUBSTRUCTUREID IS ZERO (0);
BEGIN LABEL FRUDECORP;
    DEFINE RETURN(RETURN1) =
        BEGIN
            FINDSUBSTRUCTUREID + RETURN1;
            GO TO FRUDECORP
        END #;

```

```

INTEGER I;
FOR I ← 1 STEP 1 UNTIL CURRENTNUMBEROFSTRUCTURES DO
BEGIN
  IF $OPD(STRUCTURETAB,
          CURRENTSTRUCTUREPTR+1-I,
          BIGTABPTR) = BIGTABPTR
  THEN RETURN(CURRENTSTRUCTUREPTR+1-I)
END
; ERUDECORP: END ;

BOOLEAN PROCEDURE FUNCTIONTEST(TYPE);
VALUE TYPE; INTEGER TYPE;
  COMMENT THIS PROCEDURE CHECKS IF THE FUNCTION ON THE
  TOP OF MSTACK IS OF THE SPECIFIED TYPE;
FUNCTIONTEST ←
  (IF BOOLEAN($OPD(MSTACK, $TOP, PROCEDUREPTRISINTOIDSTACK))
  THEN $OPD(IDSTACK, $OPD(MSTACK, $TOP, PROCEDUREPTR), TYPE)
  ELSE $OPD(STRUCTURETAB, $OPD(MSTACK, $TOP, PROCEDUREPTR), TYPE))
  = TYPE;

BOOLEAN PROCEDURE OFFSETEQUAL(A, LA, R);
VALUE A, LA, R; INTEGER A, LA, R;
  COMMENT THIS PROCEDURE DOES AN EQUALSTRUCTURETAB WHERE A AND
  LA POINT TO THE A VECTOR AND DESCRIBE ITS LENGTH. R IS AS IN
  EQUALSTRUCTURETAB;
BEGIN LABEL ERUDECORP;
  DEFINE RETURN(RETURN1) =
    BEGIN
      OFFSETEQUAL ← RETURN1;
      GO TO FRUDECORP
    END #;
  INTEGER J, P;
  P ← $OPD(STRUCTURETAB, R, STRUCTUREPTR);
  IF LA ≠ $OPD(STRUCTURETAB, R, NUMBERSUBSTRUCT)
  THEN BEGIN
    RETURN(FALSE)
  END ELSE BEGIN
    FOR I ← 1 STEP 1 UNTIL LA DO
      IF NOT EQUALSTRUCTURETAB(A+1-I, P+1-I)
      THEN BEGIN RETURN(FALSE) END
    END;
    RETURN(TRUE)
  ; ERUDECORP: END ;

PROCEDURE TESTEFTYPE(TYPE); VALUE TYPE; INTEGER TYPE;
  COMMENT THIS PROCEDURE PRINTS AN ERROR MESSAGE IF THE NEXT
  FORMAL PARAMETER IN A PROCEDURE DOES NOT HAVE TYPE "TYPE";
  IF CURRENTACTUALPARAMETERCOUNT ≥ CURRENTNUMBEROFFORMALPARAMETERS
  THEN
  BEGIN
    ERRMS "NUMBER OF ACTUAL PARAMETERS EXCEEDS NUMBER ALLOWED." ;
    WRITESOUT;
    SEMANTICTEST ← FALSE
  END
  ELSE IF $OPD(STRUCTURETAB, CURRENTFORMALPARAMETERLISTPOINTER -

```

```

                                CURRENTACTUALPARAMETERCOUNT,TYPE)#TYPE 00418500
THEN SEMANTICTEST ← FALSE 00418600
ELSE SEMANTICTEST ← TRUE; 00418700
                                00418800
                                00418900
                                00419000
PROCEDURE TESTACCESSTYPE(TYPE); VALUE TYPE; INTEGER TYPE; 00419100
COMMENT THIS PROCEDURE TESTS THE TYPE OF THE ACCESS VARIABLE 00419200
ON THE TOP OF MSTACK. IF IT IS FOLLOWED BY AN ARROW, IT IS TESTED 00419300
FOR A STORE TYPE; 00419400
DROP 00419500
    GETLKSVM(1); 00419600
    IF $OPD(MSTACK,$TOP+1,PSYM) = ":" 00419700
        OR $OPD(MSTACK,$TOP+1,PSYM) = "<" 00419800
    THEN SEMANTICTEST ← TYPE = 00419900
        $OPD(IDSTACK,$OPD(RIGTAR,$OPD(MSTACK,$TOP,ENTRY)), 00420000
            IDSTACKPTR),STORETYPE) 00420100
    ELSE SEMANTICTEST ← TYPE = 00420200
        $OPD(IDSTACK,$OPD(RIGTAR,$OPD(MSTACK,$TOP,ENTRY)), 00420300
            IDSTACKPTR),FETCHTYPE); 00420400
END TESTACCESSTYPE; 00420500
                                00500100
                                00500200
                                00500300
ACTION(0): 00500400
    DROP 00500500
    PMESOUT ← POINTER(MESOUT(0)); 00500600
    PTIDSTACK ← 1; 00500700
    PTSTRUCTURETAR ← 1; 00500800
    CURRENTSEGMENT ← CURRENTLEVEL + -1; 00500900
    END; 00501000
ACTION(ACCESSBOOLEAN): 00501100
    TESTACCESSTYPE(BOOL); 00501110
ACTION(ACCESSCODE): 00501200
ACTION(ACCESSDECLARATION): 00501300
    COMMENT THIS ACTION SETS UP AN ACCESS DECLARATION SCAN. 00501400
    CURRENTDECLTYPE IS SET UP AND CURRENTACCESSID IS SET TO POINT TO 00501500
    THE DECLARATION OF THE ACCESS TYPE IN IDSTACK; 00501600
    DROP 00501700
    CURRENTDECLTYPE ← ACCESSVAR; 00501800
    CURRENTACCESSID ← $OPD(RIGTAR,$OPD(MSTACK,$TOP,ENTRY),IDSTACKPTR); 00501900
    END; 00502000
ACTION(ACCESSFILE): 00502100
ACTION(ACCESSIDENTIFIER): 00502200
    COMMENT THIS ACTION ENTERS THE ID INTO IDSTACK AND SETS UP FOR 00502300
    AN ACCESS DEFINITION. IF THE ID IS IN USE IN THIS BLOCK, AN ERROR 00502400
    MESSAGE IS PRINTED; 00502500
    DROP 00502600
    CURRENTDECLTYPE ← ACCESS; 00502700
    $EXEC(ENTERIDINDECLSTACK); 00502800
    $ENTER(IDSTACK,$TOP,STRUCTPTRBEFORE:PTSTRUCTURETAR, 00502900
        ACCESTYPE: 00503000
            (CURRENTMAXACCESSNUMBER + 00503100
                CURRENTMAXACCESSNUMBER+1)); 00503200
    $PUSH(PROCEDUREIDSTACK,CURRENTPROCEDUREID); 00503300
    CURRENTPROCEDUREID ← PTIDSTACK-1; 00503400
    $EXEC(SCANSUBSTRUCTURE); 00503500
    END; 00503600
ACTION(ACCESSINTEGER): 00503700
    TESTACCESSTYPE(INT); 00503710
ACTION(ACCESSPATTERN): 00503720

```

```

    TESTACCFSTYPE(PATT);                                00503730
$ACTION(ACCFSSPOINTFR);                                00503800
    TESTACCFSTYPE(PTR);                                  00503810
$ACTION(ACCFSSPROCESS);                                00503900
$ACTION(ACCFSSSTRUCTURE);                              00504000
    TESTACCFSTYPE(STRUCT);                              00504010
$ACTION(ACCESSTRING);                                  00504100
    TESTACCFSTYPE(STR);                                  00504110
$ACTION(ADDANOTHERSUBSCRIPT);                          00504200
    $INCR(CURRENTNUMBEROFSUPS);                         00504300
$ACTION(BOOLEANDECLARATION);                          00504400
    COMMENT THIS ACTION SETS CURRENTDECLTYPE = BOOL TO INDICATE 00504500
    THAT A BOOLEAN DECLARATION IS BEING SCANNED ;          00504600
    CURRENTDECLTYPE = BOOL;                              00504700
$ACTION(BOOLEANPROCEDURE);                            00504800
    DROP                                                  00504900
    SEMANTICTEST = (IF CURRENTSTRUCTUREPTR = 0            00505000
                    THEN $OPD(IDSTACK,CURRENTPROCEDUREID,TYPE) 00505100
                    ELSE $OPD(STRUCTURETAB,CURRENTIDPTR,TYPE)) 00505200
                    = BOOL ;                                00505300
    END;                                                  00505400
$ACTION(BOOLEANSTORE);                                00505500
    DROP                                                  00505600
    OPERATOR("STORE-BOOLEAN");                          00505700
    END;                                                  00505800
$ACTION(CHECKFORMALALLOWED);                          00505900
    COMMENT THIS ACTION MAKES SURE THAT A FORMAL PROCEDURE BODY 00506000
    IS ALLOWED;                                           00506100
    DROP                                                  00506200
    IF NOT $OPD(FORMALSWITCHSTACK,$TOP-1) % CHK IF FORML PARM SCANING 00506300
    THEN                                                  00506400
    BEGIN                                                00506500
        ERRMS "A ","","FORMAL","","" SPECIFICATION IS ONLY ALLOWED " 00506600
        "AS A PROCEDURE BODY IF THE DECLARED PROCEDURE " 00506700
        "IS A FORMAL PARAMETER OF ANOTHER PROCEDURE." 00506800
        WRITEMESOUT                                     00506900
    END;                                                  00507000
    END;                                                  00507100
$ACTION(CHECKFORWARDOK);                              00507200
    COMMENT THIS ACTION CHECKS TO MAKE SURE THAT A FORWARD      00507300
    PROCEDURE BODY IS ALLOWED;                             00507400
    DROP                                                  00507500
    I = $OPD(IDSTACK,CURRENTPROCEDUREID,RIGTABPTR);      00507600
    IF $OPD(RIGTAB,I,FORWARDDEFINED) = 1                 00507700
    THEN                                                  00507800
    BEGIN                                                00507900
        ERRMS "HEY DUMMY - ONLY ONE FORWARD PER PROCEDURE IS ALLOWED." 00508000
        ; WRITEMESOUT                                     00508100
    END;                                                  00508200
    ELSE IF SCANNINGFORMALPARAMETERDECLARATIONS          00508300
    THEN                                                  00508400
    BEGIN                                                00508500
        ERRMS "FORWARD IS NOT AN ALLOWED PROCEDURE BODY HERE. ONLY " 00508600
        "FORMAL IS PERMITTED. ";                        00508700
        WRITEMESOUT                                     00508800
    END;                                                  00508900
    ELSE $OPD(RIGTAB,I,FORWARDDEFINED) = 1;              00509000
    END;                                                  00509100
$ACTION(CHECKIFNEXTFFISAPROCFDURE);                   00509200
    DROP                                                  00509300
    IF CURRENTACTUALPARAMETERCOUNT ≥ CURRENTNUMBEROFFORMALPARAMETERS 00509400

```

```

THEN                                                    00509500
BEGIN                                                    00509600
  ERRMS "NUMBER OF ACTUAL PARAMETERS EXCEEDS NUMBER ALLOWED." ; 00509700
  WRITEMESOUT;                                         00509800
  SEMANTICTEST + TRUE;                                00509900
END                                                    00510000
ELSE                                                    00510100
IF BOOLEAN($OPD(STRUCTURETAB,CURRENTFORMALPARAMETERLISTPTR - 00510200
               CURRENTACTUALPARAMETERCOUNT,PROCEDUREF)) 00510300
THEN                                                    00510400
BEGIN                                                    00510500
  SEMANTICTEST + TRUE;                                00510600
  IF NOT(IF BOOLEAN($OPD(MSTACK,$TOP,PROCEDUREPTRISINTOIDSTACK)) 00510700
        THEN EQUALIDSTRUCT($OPD(MSTACK,$TOP,PROCEDUREPTR), 00510800
                          CURRENTFORMALPARAMETERLISTPTR - 00510900
                          CURRENTACTUALPARAMETERCOUNT) 00511000
        ELSE EQUALSTRUCTURETAB($OPD(MSTACK,$TOP,PROCEDUREPTR), 00511100
                              CURRENTFORMALPARAMETERLISTPTR - 00511200
                              CURRENTACTUALPARAMETERCOUNT)) 00511300
    THEN                                                    00511400
    BEGIN                                                    00511500
      ERRMS "PROCEDURE ID DOES NOT MATCH THE PROCEDURE FORMAL", 00511600
      " PARAMETER IN THE PROCEDURE DECLARATION." ; 00511700
      WRITEMESOUT                                         00511800
    END                                                    00511900
  ELSE SEMANTICTEST + FALSE;                             00512000
END;                                                    00512100
ACTION(CHECKIFNEXTFPISACCESS);                          00512200
DROP                                                    00512300
IF CURRENTACTUALPARAMETERCOUNT ≥ CURRENTNUMBEROFFORMALPARAMETERS 00512400
THEN                                                    00512500
BEGIN                                                    00512600
  ERRMS "NUMBER OF ACTUAL PARAMETERS EXCEEDS NUMBER ALLOWED." ; 00512700
  WRITEMESOUT;                                         00512800
  SEMANTICTEST + TRUE;                                00512900
END;                                                    00513000
ELSE                                                    00513100
SEMANTICTEST + FALSE;                                00513200
END;                                                    00513300
ACTION(CHECKIFNEXTFPISBOOLEAN); TESTFPTYPE(BOOL );      00513400
ACTION(CHECKIFNEXTFPISCODE ); TESTFPTYPE(OSLCODE);      00513500
ACTION(CHECKIFNEXTFPISFVENT ); TESTFPTYPE(FVNT );       00513600
ACTION(CHECKIFNEXTFPISFILE ); TESTFPTYPE(OSLFILE);      00513700
ACTION(CHECKIFNEXTFPISINTEGER); TESTFPTYPE(INT );       00513800
ACTION(CHECKIFNEXTFPISPATTERN); TESTFPTYPE(PATT );      00513900
ACTION(CHECKIFNEXTFPISPOINTER); TESTFPTYPE(PTR );       00514000
ACTION(CHECKIFNEXTFPISPROCESS); TESTFPTYPE(PROCS );     00514100
ACTION(CHECKIFNEXTFPISSTRING ); TESTFPTYPE(STR );       00514200
ACTION(CHECKIFNEXTFPISSTRUCTURE);                      00514300
COMMENT THIS ACTION PRINTS AN ERROR MESSAGE IF THE SCANNED 00514400
ACTUAL PARAMETER WHICH WAS A STRUCTURE, IS NOT STRUCTURALLY 00514500
IDENTICAL TO THE PROCEDURE FORMAL PARAMETER OR IF THE FORMAL 00514600
PARAMETER IS NOT A STRUCTURE;                          00514700
DROP                                                    00514800
IF CURRENTACTUALPARAMETERCOUNT ≥ CURRENTNUMBEROFFORMALPARAMETERS 00514900
THEN BEGIN                                              00515000
  ERRMS "NUMBER OF ACTUAL PARAMETERS EXCEEDS NUMBER ALLOWED." ; 00515100
  WRITEMESOUT                                           00515200
END ELSE BEGIN                                          00515300
  IF $OPD(STRUCTURETAB,
          00515400
          00515500

```

```

CURRENTFORMALPARAMETERLISTPTR = 00515600
CURRENTACTUALPARAMETERCOUNT,TYPE) ≠ STRUCT 00515700
THEN BEGIN 00515800
    ERRMS "A STRUCTURE IS NOT ALLOWED AS AN ACTUAL ", 00515900
    "PARAMETER AT THIS PARAMETER POSITION."; 00516000
    WRITEMESOUT 00516100
END ELSE BEGIN 00516200
    IF NOT OFFSETEQUAL($OPD(MSTACK,$TOP,STRUCTUREPTR), 00516300
        $OPD(MSTACK,$TOP,NUMPROSUBSTRUCT), 00516400
        CURRENTFORMALPARAMETERLISTPTR + 00516500
        CURRENTACTUALPARAMETERCOUNT) 00516600
    THEN BEGIN 00516700
        ERRMS "STRUCTURE DOES NOT MATCH FORMAL PARAMETER ", 00516800
        "STRUCTURE IN PROCEDURE DECLARATION."; 00516900
        WRITEMESOUT 00517000
    END 00517100
END 00517200
END; 00517300
END; 00517400
SACTION(CHECKNOTFORMAL); 00517500
    COMMENT THIS ACTION CHECKS THAT A STATEMENT BODY TO A PROCEDURE 00517600
    DECLARATION WAS ALLOWED AND TURNS OFF THE FORWARD BIT IN BIGTAB 00517700
    IF IT WAS ON; 00517800
    DROP 00517900
    IF SCANNINGFORMALPARAMETERDECLARATIONS 00518000
    THEN 00518100
    BEGIN 00518200
        ERRMS "PROCEDURE DECLARATIONS USED TO DEFINE FORMAL ", 00518300
        "PARAMETERS IN A PROCEDURE HEADING MAY ONLY USE ", 00518400
        "","", "FORMAL","", " AS A BODY."; 00518500
        WRITEMESOUT 00518600
    END 00518700
    ELSE 00518800
    BEGIN 00518900
        $OPD(BIGTAB,$OPD(IDSTACK,CURRENTPROCEDUREID,BIGTABPTR), 00519000
            FORWARDDEFINED) ← 0; 00519100
    END; 00519200
    END; 00519300
SACTION(CHECKSUBSCRIPTEDFORUPPER); 00519400
SACTION(CHECKSUBSCRIPTMATCH); 00519500
    COMMENT THIS ACTION PRINTS AN ERROR MESSAGE IF THE NUMBER OF 00519600
    SUBSCRIPTS DOES NOT MATCH THE DECLARATION; 00519700
    DROP 00519800
    IF (IF CURRENTSTRUCTUREPTR = 0 00519900
        THEN $OPD(IDSTACK,$OPD(BIGTAB,$OPD(MSTACK,$TOP-2,ENTRY), 00520000
            IDSTACKPTR),NUMPROFSURS) 00520100
        ELSE $OPD(STRUCTURETAB,CURRENTIDPTR,NUMPROFSURS)) ≠ 00520200
        CURRENTNUMPROFSURS 00520300
    THEN 00520400
    BEGIN 00520500
        ERRMS "NUMBER OF SUBSCRIPTS DOES NOT MATCH DECLARATION."; 00520600
        WRITEMESOUT 00520700
    END; 00520800
    $PCP(SUBSCRIPTSTACK;CURRENTNUMPROFSURS); 00520900
    $PCP(SUBSCRIPTSTACK;CURRENTSTRUCTUREPTR); 00521000
    $PCP(SUBSCRIPTSTACK;CURRENTNUMBEROFSUBSTRUCTURES); 00521100
    NONAMECALL ← TRUE; 00521200
    END; 00521300
SACTION(CHECKTHATIDISTYPED); 00521400
    COMMENT THIS ACTION SETS SEMANTICTEST TRUE IF THE PROCEDURE ID 00521500
    ON THE TOP OF THE STACK IS A TYPED PROCEDURE ID; 00521600

```

```

DROP SEMANTICTEST +                                00521700
      (IF BOOLEAN($OPD(MSTACK,$TOP,PROCEDUREPTRISINTOIDSTACK)) 00521800
        THEN $OPD(IDSTACK,                                00521900
          $OPD(MSTACK,$TOP,PROCEDUREPTR),                00522000
          TYPE)                                           00522100
        ELSE $OPD(STRUCTUREPTR,                          00522200
          $OPD(MSTACK,$TOP,PROCEDUREPTR),                00522300
          TYPE)) # UNDECL;                               00522400
      00522500
END;                                                    00522600
COMMENT                                                00522700
  THE FOLLOWING ACTIONS ALL TEST IF THE FUNCTION ON THE TOP 00522800
  OF MSTACK IS OF THE INDICATED TYPE;                  00522900
$ACTION(CHECKTHATPARMSNOTREQUIRED):                    00523000
  COMMENT THIS ACTION INSURES THAT A PROCEDURE ID STANDING 00523100
  ALONE SHOULD NOT HAVE ACTUAL PARAMETERS. IF ACTUAL PARAMETERS 00523200
  ARE REQUIRED, AN ERROR MESSAGE IS PRINTED;             00523300
DROP                                                    00523400
IF (IF BOOLEAN($OPD(MSTACK,$TOP,PROCEDUREPTRISINTOIDSTACK)) 00523500
  THEN $OPD(IDSTACK,$OPD(MSTACK,$TOP,PROCEDUREPTR),NUMBROFPARMS) 00523600
  ELSE $OPD(STRUCTUREPTR,$OPD(MSTACK,$TOP,PROCEDUREPTR),
    NUMBROFPARMS)) # 0                                00523700
THEN BEGIN                                             00523800
  ERRMS "THIS PROCEDURE REQUIRES ACTUAL PARAMETERS.";   00523900
  WRITEMESOUT                                           00524000
END;                                                    00524100
END;                                                    00524200
$ACTION(CHECKTHATPROCEDUREISNOTTYPED ):              00524300
  COMMENT THIS ACTION GENERATES AN ERROR IF A RETURN IS USED 00524400
  WITHOUT AN EXPRESSION IN A TYPED PROCEDURE DECLARATION; 00524500
DROP                                                    00524600
IF $OPD(IDSTACK,CURRENTPROCEDUREID,TYPE) # UNDECL    00524700
THEN                                                    00524800
BEGIN                                                  00524900
  ERRMS "AN EXPRESSION IS EXPECTED FOLLOWING THE ",,"","RETURN", 00525000
  "","","";                                             00525100
  WRITEMESOUT;                                         00525200
END;                                                    00525300
COMMENT                                                00525400
  THE FOLLOWING ACTIONS ALL SET SEMANTICTEST TRUE        00525500
  IF THE PROCEDURE CURRENTLY BEING DECLARED IS OF THE INDICATED 00525600
  TYPE;                                                  00525700
END;                                                    00525800
$ACTION(CHECKTHATRETURNSTRUCTUREMATCHESPROCEDURESTRUCTURE): 00525900
  COMMENT IN A RETURN STATEMENT, THIS ACTION CHECKS WHETHER THE 00526000
  DECLARED PROCEDURE STRUCTURE MATCHES THE RETURNED STRUCTURE. 00526100
  IF IT DOES NOT AN ERROR MESSAGE IS PRINTED;          00526200
BEGIN                                                  00526300
  DEFINE ERROR = BEGIN ERRMS "RETURNED STRUCTURE DOES NOT ", 00526400
    "MATCH THE DECLARED PROCEDURE STRUCTURE.";          00526500
    WRITEMESOUT END #;                                  00526600
  LABEL LOOP;                                           00526700
  INTEGER PROCEDURESTRUCTUREPOINTER,                   00526800
    NUMBROFFPROCEDURESUBSTRUCTURES,                     00526900
    RETURNSTRUCTUREPOINTER,                             00527000
    NUMBROFFRETURNSUBSTRUCTURES;                       00527100
  IF CURRENTSTRUCTUREPTR = 0                            00527200
  THEN                                                  00527300
  BEGIN                                                00527400
    PROCEDURESTRUCTUREPOINTER +                         00527500
      $OPD(IDSTACK,CURRENTPROCEDUREID,STRUCTUREPTR); 00527600

```

```

NUMBEROFPROCEDURESUBSTRUCTURES + 00527800
      $OPD(IDSTACK,CURRENTPROCEDUREID,NUMBRSUBSTRUCT) 00527900
END 00528000
ELSE 00528100
BEGIN 00528200
      PROCEDURESTRUCTUREPOINTER + 00528300
      $OPD(STRUCTURETAB,CURRENTIDPTR,STRUCTUREPTR); 00528400
      NUMBEROFPROCEDURESUBSTRUCTURES + 00528500
      $OPD(STRUCTURETAB,CURRENTIDPTR,NUMBRSUBSTRUCT) 00528600
END; 00528700
RETURNSTRUCTUREPOINTER + $OPD(MSTACK,$TOP,STRUCTUREPTR); 00528800
NUMBEROFRETURNSUBSTRUCTURES+$OPD(MSTACK,$TOP,NUMBRSUBSTRUCT); 00528900
IF NUMBEROFPROCEDURESUBSTRUCTURES = 00529000
      NUMBEROFRETURNSUBSTRUCTURES 00529100
THEN 00529200
      BEGIN 00529300
      LOOP: 00529400
            FOR I + 1 STEP 1 UNTIL NUMBEROFRETURNSUBSTRUCTURES DO 00529500
            IF NOT EQUALSTRUCTURETAB 00529600
                  (PROCEDURESTRUCTUREPOINTER+1-I, 00529700
                  RETURNSTRUCTUREPOINTER+1-I) 00529800
            THEN 00529900
            BEGIN 00530000
                  ERROR; 00530100
                  LEAVE LOOP 00530200
            END; LOOP; 00530300
      END 00530400
END; 00530500
COMMENT 00530600
      THESE FOLLOWING ACTIONS ALL CHECK THAT AN ID IS OF THE 00530700
      INDICATED TYPE. IF CURRENTSTRUCTUREPTR = 0 WE GO RIGHT TO IDSTACK 00530800
      FROM BIGTAB. OTHERWISE WE TRY TO FIND THE ID IN STRUCTURETAB IN 00530900
      THE CURRENT AREA OF THE SUBSTRUCTURE; 00531000
ACTION( CHECKTHATSTRUCTURESARETHESAME ); 00531100
      COMMENT THIS ACTION IS INVOKED IN A STRUCTURE ASSIGNMENT TO 00531200
      INSURE THAT THE STRUCTURES ON THE LEFT AND RIGHT OF THE ASSIGNMENT 00531300
      ARROW ARE IDENTICALLY TYPED AND IDENTICALLY STRUCTURED ; 00531400
      BEGIN 00531500
            INTEGER RIGHTPTR,RIGHTNUMBER,LEFTPTR,LEFTNUMBER; 00531600
            LABEL LOOP; 00531700
            DEFINE ERROR = BEGIN ERRMS "STRUCTURES DO NOT MATCH." ; 00531800
                  WRITMESOUT FND #; 00531900
            RIGHTPTR + $OPD(MSTACK,$TOP,STRUCTUREPTR); 00532000
            RIGHTNUMBER + $OPD(MSTACK,$TOP,NUMBRSUBSTRUCT); 00532100
            LEFTPTR + $OPD(MSTACK,$TOP-2,STRUCTUREPTR); 00532200
            LEFTNUMBER + $OPD(MSTACK,$TOP-2,NUMBRSUBSTRUCT); 00532300
            IF LEFTNUMBER = RIGHTNUMBER 00532400
            THEN 00532500
            BEGIN 00532600
            LOOP: 00532700
                  FOR I + 1 STEP 1 UNTIL LEFTNUMBER DO 00532800
                  BEGIN 00532900
                        IF NOT EQUALSTRUCTURETAB(LEFTPTR+1-I,RIGHTPTR+1-I) 00533000
                        THEN 00533100
                        BEGIN 00533200
                              ERROR; 00533300
                              LEAVE LOOP 00533400
                        END 00533500
                        FND; LOOP; 00533600
                  END 00533700
            END 00533800
            ELSE ERROR 00533900

```

```

END;                                00533900
&ACTION(CLOSELABEL)                00534000
    COMMENT THIS ACTION REMOVES A LABEL FROM THE IDSTACK. IF THIS 00534100
    LABEL IS NOT THE RIGHT ONE, IT TRIES TO FIX IT UP;           00534200
DROP                                00534300
IF $OPD(MSTACK,$TOP,ENTRY) = $OPD(IDSTACK,$TOP,BIGTARPTR)      00534400
    AND $OPD(IDSTACK,$TOP,TYPE) = LAPL                             00534500
THEN                                00534600
ELSE                                00534700
BEGIN                               00534800
    ERRMS "THIS CLOSING LABEL DOES NOT MATCH THE OPENING LABEL. "; 00534900
    WRITEMESOUT;                                                  00535000
END;                                00535100
WHILE $OPD(LEAVESTACK,$TOP,MARK) # 1 DO                          00535200
BEGIN                               00535300
    BRANCH($OPD(LEAVESTACK,$TOP,ADDRESS),                          00535400
        CURRENTADDRESS,"BRANCH");                                00535500
    $POP(LEAVESTACK);                                              00535600
END;                                00535700
$POP(LEAVESTACK);                                                  00535800
$EXEC(ENDOFBLOCK);                                                00535900
END;                                00536000
&ACTION(CODEARS):                                                    00600100
DROP                                00600200
OPERATOR("ARS");                                                  00600300
END;                                00600400
&ACTION(CODEACCESSDECLARATION):                                       00600500
&ACTION(CODEACCESSIDACTUALPARAMETER):                                00600600
&ACTION(CODEALLOCATEFUNCTION):                                         00600700
&ACTION(CODEALLOCATEPROCEDURE):                                        00600800
&ACTION(CODEAND):                                                      00600900
DROP                                00601000
OPERATOR("AND");                                                  00601100
END;                                00601200
&ACTION(CODEANY):                                                      00601300
&ACTION(CODEANYDEFAULT):                                               00601400
&ACTION(CODEBEGINAPPEND):                                              00601500
&ACTION(CODEBEGINBOOLEANINITIALIZATION):                              00601600
    $EXEC(CODEBEGININTEGERINITIALIZATION);                          00601700
&ACTION(CODEBEGINCAUSE):                                               00601800
&ACTION(CODEBEGINCONSTANTSTATEMENT):                                  00601900
&ACTION(CODEBEGINEVENTACTIONDECL):                                     00602000
&ACTION(CODEBEGINFORBY):                                               00602100
&ACTION(CODEBEGINFORLIMIT):                                            00602200
&ACTION(CODEBEGINFORSTEP):                                             00602300
&ACTION(CODEBEGINFORUNTIL):                                            00602400
&ACTION(CODEBEGINFORWHILE):                                            00602500
&ACTION(CODEBEGININITIATE):                                            00602600
&ACTION(CODEBEGININTEGERINITIALIZATION):                              00602700
BEGIN                               00602800
    INTEGER I,J;                                                    00602900
    I ← CURRENTMAXIMUMDISPLACEMENT - IDLISTCOUNTER;                00603000
    J ← I+IDLISTCOUNTER-1;                                           00603100
    FOR I ← I STEP 1 UNTIL J DO                                       00603200
        COUPLE("NAME-CALL",CURRENTLEVEL,I," ");                  00603300
    END;                                                              00603400
&ACTION(CODEBEGINLOWERBOUND):                                          00603500
&ACTION(CODEBEGINPATTERNINITIALIZATION):                              00603600
&ACTION(CODEBEGINPOINTINTERINITIALIZATION):                          00603700
    $EXEC(CODEBEGININTEGERINITIALIZATION);                          00603800
&ACTION(CODEBEGINPROCEDUREBODY):                                       00603900

```

\$ACTION(CODEBEGINPROCESSTERMINATE):	00604000
\$ACTION(CODEBEGINRESTART):	00604100
\$ACTION(CODEBEGINSPAWN):	00604200
\$ACTION(CODEBEGINSTRINGINITIALIZATION):	00604300
\$EXFC(CODEBEGININTEGERINITIALIZATION);	00604400
\$ACTION(CODEBEGINSTRINGLENGTH):	00604500
\$ACTION(CODEBEGINSTRINGSIZE):	00604600
\$ACTION(CODEBEGINSTRUCTUREDECL):	00604700
\$ACTION(CODEBEGIN'SUSPEND):	00604800
\$ACTION(CODEBEGINUPPERROUND):	00604900
\$ACTION(CODEBEGINVALUE):	00605000
\$ACTION(CODEBEGINWAIT):	00605100
\$ACTION(CODEBEGINWHILESTATEMENT):	00605200
\$ACTION(CODEBLOCKEND):	00605300
DROP	00605400
\$POP(BLOCKSTACK,CURRENTMAXIMUMDISPLACEMENT);	00605500
\$POP(BLOCKSTACK,CURRENTADDRESS);	00605600
\$POP(BLOCKSTACK,CURRENTSEGMENT);	00605700
\$DECR(CURRENTLEVEL);	00605800
END;	00605900
\$ACTION(CODEBOOLEANACTUALPARAMETER):	00606000
\$ACTION(CODEBOOLEANFUNCTION):	00606100
\$ACTION(CODEBOOLEANVALUECALL):	00606200
DROP	00606300
OPERATOR("LOAD-BOOLEAN");	00606400
END;	00606500
\$ACTION(CODEBOOLEANVARIABLE):	00606600
DROP	00606700
IF NOT NONAMECALL THEN	00606800
COUPLE("NAME-CALL",\$OPD(IDSTACK,I + \$OPD(BIGTAB,	00606900
\$OPD(MSTACK,STOP,ENTRY),IDSTACKPTR),LEVEL),	00607000
\$OPD(IDSTACK,I,DISPLACEMENT),PRINTID);	00607100
LASTTYPE + BOOL;	00607200
NONAMECALL + FALSE;	00607300
END;	00607400
\$ACTION(CODEBRANCHBACKFOR):	00607500
\$ACTION(CODEBRANCHBACKFORTIMES):	00607600
\$ACTION(CODEBRANCHBACKTODIFFALSE):	00607700
\$ACTION(CODEBRANCHBACKTOWHILE):	00607800
\$ACTION(CODEBRANCHPASTELSE):	00607900
\$ACTION(CODEBRANCHPASTSTATEMENTIFFALSE):	00608000
\$ACTION(CODEBRANCHPASTTHENIFFALSE):	00608100
\$ACTION(CODECASEBRANCH):	00608200
\$ACTION(CODECASESTATEMENTENTRY):	00608300
\$ACTION(CODECASEWAITSTATEMENT):	00608400
\$ACTION(CODECAUSEFTERMINATE):	00608500
\$ACTION(CODECDIV):	00608600
DROP	00608700
OPERATOR("CDIV");	00608800
END;	00608900
\$ACTION(CODECHARSIZEOFSTRING):	00609000
DROP	00609100
OPERATOR("CHARSIZE");	00609200
END;	00609300
\$ACTION(CODECODEACTUALPARAMETER):	00609400
SEMANTICTEST + FALSE;	00609500
\$ACTION(CODECONVERT):	00609600
DROP	00609700
OPERATOR("TRANSLATE");	00609800
END;	00609900
\$ACTION(CODEDECLARATION):	00610000

```

      COMMENT THIS ACTION SETS CURRENTDECLTYPE = OSLCODE TO INDICATE THAT A CODE DECLARATION IS BEING SCANNED ;
      CURRENTDECLTYPE ← OSLCODE;
$ACTION(CODEDECTOINTEGER);
  DROP
  OPERATOR("CONVERTDECIMALTOINTEGER");
  END;
$ACTION(CODEDEFAULTACCESSIDACTUALPARAMETER);
$ACTION(CODEDEFAULTBOOLEANACTUALPARAMETER);
$ACTION(CODEDEFAULTCODEACTUALPARAMETER);
$ACTION(CODEDEFAULTEVENTACTUALPARAMETER);
$ACTION(CODEDEFAULTFILEACTUALPARAMETER);
$ACTION(CODEDEFAULTFORINCREMENT);
$ACTION(CODEDEFAULTINTEGERACTUALPARAMETER);
$ACTION(CODEDEFAULTPATTERNACTUALPARAMETER);
$ACTION(CODEDEFAULTPOINTINTERACTUALPARAMETER);
$ACTION(CODEDEFAULTPROCEDUREIDACTUALPARAMETER);
$ACTION(CODEDEFAULTPROCESSACTUALPARAMETER);
$ACTION(CODEDEFAULTSTRINGACTUALPARAMETER);
$ACTION(CODEDEFAULTSTRUCTUREACTUALPARAMETER);
$ACTION(CODEDEFAULTTERMINATE);
$ACTION(CODEDIVIDE);
  DROP
  OPERATOR("DIVIDE");
  END;
$ACTION(CODEENDAPPEND);
$ACTION(CODEENDBOOLEANINITIALIZATION);
  BEGIN
    INTEGER I;
    FOR I ← 1 STEP 1 UNTIL IDLISTCOUNTER-1 DO
      OPERATOR("STORE-BOOLEAN-PARTIAL-DESTRUCTIVE");
      OPERATOR("STORE-BOOLEAN");
    END;
$ACTION(CODEENDBRANCHIFFALSE);
$ACTION(CODEENDBRANCHPASTELSE);
$ACTION(CODEENDCAUSE);
$ACTION(CODEENDCAUSETERMINATE);
$ACTION(CODEENDEVENTACTIONDECL);
$ACTION(CODEENDFALSEBRANCHPASTTHEN);
$ACTION(CODEENDFORBY);
$ACTION(CODEENDFORLIMIT);
$ACTION(CODEENDFORLIST);
$ACTION(CODEENDFORSTEP);
$ACTION(CODEENDFORUNTIL);
$ACTION(CODEENDFORWHILE);
$ACTION(CODEENDINITIATE);
$ACTION(CODEENDINTEGERINITIALIZATION);
  BEGIN
    INTEGER I;
    FOR I ← 1 STEP 1 UNTIL IDLISTCOUNTER-1 DO
      OPERATOR("STORE-INTEGER-PARTIAL-DESTRUCTIVE");
      OPERATOR("STORE-INTEGER");
    END;
$ACTION(CODEENDMATCHASSIGNMENT);
$ACTION(CODEENDOFFCASE);
$ACTION(CODEENDOFFCASEWAIT);
$ACTION(CODEENDOFFEXPCASE);
  DROP
  FOR I ← CURRENTCASECOUNT STEP -1 UNTIL 1 DO
  BEGIN
    BRANCH(CURRENTADDRESS,$OPD(CASESTACK,$TOP-(2*I)),"BRANCH");

```

```

    $INCR(CURRENTADDRESS);
    BRANCH($OFD(CASESTACK,$TOP+1-(2*I)),
        $OPD(CASESTACK,$TOP)+CURRENTCASECOUNT,"BRANCH");
END;
BRANCH($OPD(CASESTACK,$TOP-1-(2*CURRENTCASECOUNT)),
    $OPD(CASESTACK,$TOP),"INDEXED-BRANCH");
PTCASESTACK ← PTCASESTACK - (2*CURRENTCASECOUNT+3);
CURRENTCASECOUNT ← $OPD(CASESTACK,$TOP+1);
END;
$ACTION(CODEENDPATTERNINITIALIZATION);
$ACTION(CODEENDPOINTINTERINITIALIZATION);
BEGIN
    INTEGER I;
    FOR J ← 1 STEP 1 UNTIL IDLISTCOUNTER-1 DO
        OPERATOR("STORE-POINTER-PARTIAL-DESTRUCTIVE");
        OPERATOR("STORE-POINTER");
    END;
$ACTION(CODEENDPROCEDUREBODY);
$ACTION(CODEENDPROCEDURESTERMINATE);
$ACTION(CODEENDRESTART);
$ACTION(CODEENDSKIPFACTOR);
$ACTION(CODEENDSPAWN);
$ACTION(CODEENDSTRINGINITIALIZATION);
BEGIN
    INTEGER I;
    FOR I ← 1 STEP 1 UNTIL IDLISTCOUNTER-1 DO
        OPERATOR("STORE-INTEGER-PARTIAL-DESTRUCTIVE");
        OPERATOR("STORE-INTEGER");
    END;
$ACTION(CODEENDSTRUCTUREDDECL);
$ACTION(CODEENDSUSPEND);
$ACTION(CODEENDVALUE);
$ACTION(CODEENDWAIT);
$ACTION(CODEEND2NDARYSKIP);
$ACTION(CODEEQL);
    RELOP ← 3;
$ACTION(CODEEQV);
    DROP
    OPERATOR("EQV");
END;
$ACTION(CODEEVENTACTUALPARAMETER);
$ACTION(CODEEVENTLIST);
$ACTION(CODEEXCEPT);
$ACTION(CODEEXCEPTDEFAULT);
$ACTION(CODEEXPCASEBRANCH);
    DROP
    $PUSH(CASESTACK,CURRENTCASECOUNT);
    $PUSH(CASESTACK,CURRENTADDRESS);
    OPERATOR("*** (INDEXED-BRANCH INSTRUCTION WILL BE HERE) ***");
    CURRENTCASECOUNT ← 0;
    $PUSH(CASESTACK,CURRENTADDRESS);
END;
$ACTION(CODEEXPCASEENTRY);
    DROP
    $PUSH(CASESTACK,CURRENTADDRESS);
    OPERATOR("*** (BRANCH INSTRUCTION WILL BE HERE) ***");
    $PUSH(CASESTACK,CURRENTADDRESS);
    $INCR(CURRENTCASECOUNT);
END;
$ACTION(CODEEXPSKIPLESE);
    DROP

```

00616200
 00616300
 00616400
 00616500
 00616600
 00616700
 00616800
 00616900
 00617000
 00617100
 00617200
 00617300
 00617400
 00617500
 00617600
 00617700
 00617800
 00617900
 00618000
 00618100
 00618200
 00618300
 00618400
 00618500
 00618600
 00618700
 00618800
 00618900
 00619000
 00619100
 00619200
 00619300
 00619400
 00619500
 00619600
 00619700
 00619800
 00619900
 00620000
 00620100
 00620200
 00620300
 00620400
 00620500
 00620600
 00620700
 00620800
 00620900
 00621000
 00621100
 00621200
 00621300
 00621400
 00621500
 00621600
 00621700
 00621800
 00621900
 00622000
 00622100
 00622200

BRANCH(\$OPD(EXPRSNIFSTACK,\$TOP),CURRENTADDRESS+1,	00622300
"BRANCH-IF-FALSE");	00622400
\$OPD(EXPRSNIFSTACK,\$TOP) + CURRENTADDRESS;	00622500
OPERATOR("*** (BRANCH INSTRUCTION WILL BE HERE) ***");	00622600
END;	00622700
\$ACTION(CODEEXPSKIPPASTELSEFOUND);	00622800
DROP	00622900
BRANCH(\$OPD(EXPRSNIFSTACK,\$TOP),CURRENTADDRESS,"BRANCH");	00623000
\$OPD(EXPRSNIFSTACK);	00623100
END;	00623200
\$ACTION(CODEEXPSKIPTHENONFALSE);	00623300
DROP	00623400
\$PUSH(EXPRSNIFSTACK,CURRENTADDRESS);	00623500
OPERATOR("*** (BRANCH-IF-FALSE INSTRUCTION WILL BE HERE) ***");	00623600
END;	00623700
\$ACTION(CODEEXPSKIPTOELSEFOUND);	00623800
\$ACTION(CODEFALSE);	00623900
DROP	00624000
OPERATOR("FALSE");	00624100
END;	00624200
\$ACTION(CODEFIELDLENGTH);	00624300
\$ACTION(CODEFIELDSTART);	00624400
\$ACTION(CODEFIELDACTUALPARAMETER);	00624500
\$ACTION(CODEFIXEDMATCH);	00624600
\$ACTION(CODEFLOATINGMATCH);	00624700
\$ACTION(CODEFORCOUNTER);	00624800
\$ACTION(CODEFORLISTEXP);	00624900
\$ACTION(CODEFORVARIABLE);	00625000
\$ACTION(CODEGEQ);	00625100
RELOP + 4;	00625200
\$ACTION(CODEGTR);	00625300
RELOP + 5;	00625400
\$ACTION(CODEIMP);	00625500
DROP	00625600
OPERATOR("IMP");	00625700
END;	00625800
\$ACTION(CODEINITIALFORASSIGNMENT);	00625900
\$ACTION(CODEINTEGERACTUALPARAMETER);	00626000
\$ACTION(CODEINTEGERADD);	00626100
DROP	00626200
OPERATOR("ADD");	00626300
END;	00626400
\$ACTION(CODEINTEGERCONSTANT);	00626500
DROP	00626600
OPERFIRSTPART,"INTEGER-LITERAL-CALL ",	00626700
PRINTID;	00626800
OPERLASTPART;	00626900
END;	00627000
\$ACTION(CODEINTEGERFUNCTION);	00627100
\$ACTION(CODEINTEGERMULTIPLY);	00627200
DROP;	00627300
OPERATOR("MULTIPLY");	00627400
END;	00627500
\$ACTION(CODEINTEGERRELOP);	00627600
OUTPUTRELOP("INTEGER");	00627700
\$ACTION(CODEINTegersubtract);	00627800
DROP	00627900
OPERATOR("SUBTRACT");	00628000
END;	00628100
\$ACTION(CODEINTEGERVALUECALL);	00628200
DROP	00628300

OPERATOR("LOAD-INTEGER");	00628400
END;	00628500
\$ACTION(CODEINTEGERVARIABLE):	00628600
DROP	00628700
IF NOT NONAMECALL THEN	00628800
COUPLE("NAME-CALL", \$OPD(IDSTACK, I + \$OPD(BIGTAB,	00628900
\$OPD(MSTACK, \$TOP, ENTRY), IDSTACKPTR), LEVEL),	00629000
\$OPD(IDSTACK, I, DISPLACEMENT), PRINTID);	00629100
LASTTYPE + INT;	00629200
NONAMECALL + FALSE;	00629300
END;	00629400
\$ACTION(CODEINTGFROMSTRING):	00629500
DROP	00629600
OPERATOR("CONVERTSTRINGTOINTEGER");	00629700
END;	00629800
\$ACTION(CODEINTGRTOBOOL):	00629900
DROP	00630000
OPERATOR("CONVERTINTEGERTOBOOLEAN");	00630100
END;	00630200
\$ACTION(CODEINTGRTODECIMAL):	00630300
DROP	00630400
OPERATOR("CONVERTINTEGERTODECIMAL");	00630500
END;	00630600
\$ACTION(CODEINTGRTOSTRING):	00630700
DROP	00630800
OPERATOR("CONVERTINTEGERTOSTRING");	00630900
END;	00631000
\$ACTION(CODELENGTHOFSTRING):	00631100
DROP	00631200
OPERATOR("LENGTH");	00631300
END;	00631400
\$ACTION(CODELEQ):	00631500
RELOP + 2;	00631600
\$ACTION(CODELONETERMINATE):	00631700
\$ACTION(CODELOWER):	00631800
\$ACTION(CODELOWERBOUND):	00631900
\$ACTION(CODELSS):	00632000
RELOP + 1;	00632100
\$ACTION(CODEMATCHASSIGNMENT):	00632200
\$ACTION(CODEMOD):	00632300
DROP	00632400
OPERATOR("MOD");	00632500
END;	00632600
\$ACTION(CODEMT):	00632700
DROP	00632800
OPERATOR("MT");	00632900
END;	00633000
\$ACTION(CODENEGATIVE):	00633100
DROP	00633200
OPERATOR("NEGATE");	00633300
END;	00633400
\$ACTION(CODENEG):	00633500
RELOP + 6;	00633600
\$ACTION(CODENONMATCHASSIGNMENT):	00633700
\$ACTION(CODENONDESTRUCTBOOLEANSTORE):	00633800
DROP	00633900
\$DECR(CURRENTADDRESS);	00634000
XOPERATOR("STORE-BOOLEAN-PARTIAL-DESTRUCTIVE");	00634100
END;	00634200
\$ACTION(CODENONDESTRUCTINTEGERSTORE):	00634300
DROP	00634400

\$DECR(CURRENTADDRESS);	00634500
XOPERATOR("STORE-INTEGER-PARTIAL-DESTRUCTIVE");	00634600
END;	00634700
\$ACTION(CODENONDESTRUCTSTRINGSTORE);	00634800
DROP	00634900
\$DECR(CURRENTADDRESS);	00635000
XOPERATOR("STORE-STRING-PARTIAL-DESTRUCTIVE");	00635100
END;	00635200
\$ACTION(CODENOSTEPFORCONDITIONAL);	00635300
\$ACTION(CODENOT);	00635400
DROP	00635500
OPERATOR("NOT");	00635600
END;	00635700
\$ACTION(CODENULL);	00635800
DROP	00635900
OPERATOR("NULL");	00636000
END;	00636100
\$ACTION(CODEOR);	00636200
DROP	00636300
OPERATOR("OR");	00636400
END;	00636500
\$ACTION(CODEPATTERNACTUALPARAMETER);	00636600
\$ACTION(CODEPATTERNMATCH);	00636700
\$ACTION(CODEPATTERNVARIABLE);	00636800
DROP	00636900
COUPLE("NAME-CALL", \$OPD(IDSTACK, I + \$OPD(BIGTAB,	00637000
\$OPD(MSTACK, \$TOP, ENTRY), IDSTACKPTR), LEVEL);	00637100
\$OPD(IDSTACK, I, DISPLACEMENT), PRINTID);	00637200
END;	00637300
\$ACTION(CODEPOINTERACTUALPARAMETER);	00637400
\$ACTION(CODEPOINTERFUNCTION);	00637500
\$ACTION(CODEPOINTERRELOP);	00637600
OUTPUTRELOP("POINTER");	00637700
\$ACTION(CODEPOINTERVALUECALL);	00637800
DROP	00637900
OPERATOR("LOAD-POINTER");	00638000
END;	00638100
\$ACTION(CODEPOINTERVARIABLE);	00638200
DROP	00638300
COUPLE("NAME-CALL", \$OPD(IDSTACK, I + \$OPD(BIGTAB,	00638400
\$OPD(MSTACK, \$TOP, ENTRY), IDSTACKPTR), LEVEL);	00638500
\$OPD(IDSTACK, I, DISPLACEMENT), PRINTID);	00638600
END;	00638700
\$ACTION(CODEPOR);	00638800
\$ACTION(CODEPOSTDECLARATIONBLOCKHEAD);	00638900
\$ACTION(CODEPOWER);	00639000
DROP	00639100
OPERATOR("POWER");	00639200
END;	00639300
\$ACTION(CODEPREDECLARATIONBLOCKHEAD);	00639400
DROP	00639500
\$PUSH(BLOCKSTACK, CURRENTSEGMENT);	00639600
\$PUSH(BLOCKSTACK, CURRENTADDRESS);	00639700
\$PUSH(BLOCKSTACK, CURRENTMAXIMUMDISPLACEMENT);	00639800
CURRENTSEGMENT + (MAXIMUMSEGMENT + MAXIMUMSEGMENT + 1);	00639900
CURRENTADDRESS + CURRENTMAXIMUMDISPLACEMENT + 0;	00640000
\$INCR(CURRENTLEVEL);	00640100
END;	00640200
\$ACTION(CODEPROCEDUREIDACTUALPARAMETER);	00640300
\$ACTION(CODEPROCESSACTUALPARAMETER);	00640400
\$ACTION(CODEPROCESSDECLARATION);	00640500

\$ACTION(CODEPTRSUBSCRIPTED);	00640600
\$ACTION(CODEPTRUNSUBSCRIPTED);	00640700
\$ACTION(CODERDIV);	00640800
DROP	00640900
OPERATOR("RDIV");	00641000
END;	00641100
\$ACTION(CODERELEASE);	00641200
\$ACTION(CODERETURNBOOLEAN);	00641300
\$ACTION(CODERETURNINTEGER);	00641400
\$ACTION(CODERETURNPATTERN);	00641500
\$ACTION(CODERETURNPOINTER);	00641600
\$ACTION(CODERETURNSTRING);	00641700
\$ACTION(CODESAND);	00641800
DROP	00641900
OPERATOR("SAND");	00642000
END;	00642100
\$ACTION(CODESIGN);	00642200
DROP	00642300
OPERATOR("SIGN");	00642400
END;	00642500
\$ACTION(CODESINGLEFORASSIGNMENT);	00642600
\$ACTION(CODESKIPFACTORIFTRUE);	00642700
\$ACTION(CODESKIP2NDARYIFFALSE);	00642800
\$ACTION(CODESNOT);	00642900
DROP	00643000
OPERATOR("SNOT");	00643100
END;	00643200
\$ACTION(CODESOR);	00643300
DROP	00643400
OPERATOR("SOR");	00643500
END;	00643600
\$ACTION(CODESPAN);	00643700
\$ACTION(CODESPANDEFAULT);	00643800
\$ACTION(CODESTRINGACTUALPARAMETER);	00643900
\$ACTION(CODESTRINGCONSTANT);	00644000
DROP	00644100
OPERFIRSTPART,"STRING-LITERAL-CALL ", "",	00644200
PRINTID,"";OPERLASTPART;	00644300
END;	00644400
\$ACTION(CODESTRINGDEFAULT);	00644500
\$ACTION(CODESTRINGFUNCTION);	00644600
\$ACTION(CODESTRINGRELOP);	00644700
OUTPUTRELOP("STRING");	00644800
\$ACTION(CODESTRINGVALUFCALL);	00644900
DROP	00645000
OPERATOR("LOAD-STRING");	00645100
END;	00645200
\$ACTION(CODESTRINGVARIABLE);	00645300
DROP	00645400
IF NOT NONAMECALL THEN	00645500
COUPLE("NAME-CALL", \$OPD(IDSTACK,I + \$OPD(BIGTAB,	00645600
\$OPD(HSTACK,STOP,ENTRY),IDSTACKPTR),LEVEL),	00645700
\$OPD(IDSTACK,I,DISPLACEMENT),PRINTID);	00645800
LASTTYPE + STR;	00645900
NONAMECALL + FALSE;	00646000
END;	00646100
\$ACTION(CODESTRUCTUREACTUALPARAMETER);	00646200
\$ACTION(CODESXOR);	00646300
DROP	00646400
OPERATOR("SXOR");	00646500
END;	00646600

```

$ACTION(CODETRUE);                                00646700
  DROP                                             00646800
  OPERATOR("TRUE");                               00646900
END;                                              00647000
$ACTION(CODEUPPER);                                00647100
$ACTION(CODEUPPERBOUND);                          00647200
$ACTION(CODEXOR);                                  00647300
  DROP                                             00647400
  OPERATOR("XOR");                                00647500
END;                                              00647600
$ACTION(CURRENTACCESSID);                          00700100
  COMMENT THIS ACTION PRINTS AN ERROR MESSAGE IF THE ID JUST
  SCANNED WAS NOT THE SAME AS THE ACCESS TYPE ID OF THE ACCESS TYPE
  CURRENTLY BEING DEFINED;                        00700200
  DROP                                             00700300
  IF $OPD(BIGTAB,$OPD(MSTACK,$TOP,ENTRY),IDSTACKPTR)
    # CURRENTPROCEDUREID                         00700400
  THEN                                           00700500
  BEGIN                                           00700600
    ERRMS "THIS IDENTIFIER SHOULD HAVE BEEN ", "",
    POINTER ($OPD(BIGTAB,(J+$OPD(IDSTACK,
    CURRENTPROCEDUREID,
    BIGTABPTR)+1),$ALL))                        00700700
    FOR $OPD(BIGTAB,I-1,IDLENGTH), "", ". ";    00700800
    WRITEMESOUT                                  00700900
  END;                                           00701000
  END;                                           00701100
  END;                                           00701200
  END;                                           00701300
  END;                                           00701400
  END;                                           00701500
  END;                                           00701600
  END;                                           00701700
  END;                                           00701800
$ACTION(ENDOFACCESSBODY);                          00701900
  COMMENT THIS ACTION ENTERS THE ACCESS PROCEDURES IN AN ARRAY IN
  STRUCTURETAB AND POINTS THE STRUCTURE POINTER AND COUNTER FIELDS AT
  THE LIST. FURTHERMORE, THE FETCH AND STORE TYPES ARE SET AND THE
  PROCEDURES POINTED TO (IN STRUCTURETAB) FOR DOING FETCHES AND STORES;
  DROP                                             00702000
  FOR I + 1 STEP 1 UNTIL IDCOUNTER DO            00702100
  BEGIN                                           00702200
    IF BOOLEAN($OPD(IDSTACK,$TOP+1-I,ACCESSPROCEDURE))
    THEN                                           00702300
    BEGIN                                           00702400
      $PUSH(STRUCTURETAB,$ALL OF (IDSTACK,$TOP+1-I));
      $INCR($OPD(IDSTACK,CURRENTPROCEDUREID,
      NUMBERSUBSTRUCT));                         00702500
      IF POINTER($OPD(BIGTAB,
      (J + $OPD(IDSTACK,$TOP+1-I,
      BIGTABPTR)+1),$ALL))
      = "FETCH" AND $OPD(BIGTAB,J-1,IDLENGTH) = 5
      THEN                                         00702600
      BEGIN                                         00702700
        IF $OPD(STRUCTURETAB,$TOP,NUMROFPARMS) = 1
        THEN                                       00702800
        BEGIN                                       00702900
          $ENTER(IDSTACK,CURRENTPROCEDUREID,
          FETCHPROCEDURE: PTSTRUCTURETAB-1,
          FETCHTYPE:
          $OPD(STRUCTURETAB,$TOP,TYPE))
          END                                     00703000
        ELSE                                     00703100
        BEGIN                                     00703200
          ERRMS "A FETCH PROCEDURE MAY HAVE ",
          "NO PARAMETERS.";
          WRITEMESOUT
        END                                     00703300
      END                                     00703400
    END                                     00703500
  END                                     00703600
  END                                     00703700
  END                                     00703800
  END                                     00703900
  END                                     00704000
  END                                     00704100
  END                                     00704200
  END                                     00704300
  END                                     00704400
  END                                     00704500
  END                                     00704600
  END                                     00704700
  END                                     00704800
  END                                     00704900
  END                                     00705000
  END                                     00705100

```

```

END                                                    00705200
ELSEF                                                  00705300
IF POINTER($OPD(BIGTAB,J,$ALL)) = "STORE"            00705400
AND $OPD(BIGTAB,J-1,IDLENGTH) = 5                    00705500
THEN                                                  00705600
BEGIN                                                00705700
    IF $OPD(STRUCTURETAB,$TOP,NUMBROFPARMS) = 2      00705800
    THEN                                              00705900
    BEGIN                                            00706000
        $ENTER(IDSTACK,CURRENTPROCEDUREID,          00706100
                STOREPROCEDURE: PTSTRUCTURETAB-1,    00706200
                STORETYPE: $OPD(STRUCTURETAB,        00706300
                                $OPD(STRUCTURETAB,    00706400
                                    $TOP,              00706500
                                    FORMALPARMPTR)+1,  00706600
                                    TYPE))             00706700
    END                                              00706800
    ELSE                                            00706900
    BEGIN                                            00707000
        ERRMS "STORE PROCEDURES ARE ALLOWED ",      00707100
        "TO HAVE AND MUST HAVE EXACTLY ",          00707200
        "ONE FORMAL PARAMETER.";                  00707300
        WRITEMESOUT                                00707400
    END                                            00707500
END                                                00707600
END;                                              00707700
END;                                              00707800
$OPD(IDSTACK,CURRENTPROCEDUREID,STRUCTUREPTR) ←      00707900
PTSTRUCTURETAB-1;                                00708000
$EXEC(ENDOFFPROCEDUREBODY);                      00708010
END;                                              00708100
$ACTION(ENDOFACCESSHEADING);                      00708200
COMMENT THIS ACTION CLEANS UP AFTER AN ACCESS HEADING. 00708300
A CHECK IS MADE TO INSURE THAT ALL FORMAL PARAMETERS WERE DECLARED; 00708400
DROP                                              00708500
FOR I ← 1 STEP 1 UNTIL IDCOUNTER DO                00708600
BEGIN                                              00708700
    $PUSH(STRUCTURETAB,$ALLOF(IDSTACK,$TOP+1-I));    00708800
    IF NOT BOOLEAN($OPD(IDSTACK,$TOP,HASBEENSPEC))    00708900
    THEN                                              00709000
    BEGIN                                            00709100
        ERRMS "FORMAL PARAMETER ", POINTER($OPD(BIGTAB,  00709200
            $OPD(IDSTACK,$TOP,BIGTARPTR)+1,$ALL))FOR    00709300
            $OPD(BIGTAB,$OPD(IDSTACK,$TOP,BIGTARPTR),    00709400
                IDLENGTH),                              00709500
            " NOT DECLARED IN ACCESS HEADING.";        00709600
        WRITEMESOUT                                00709700
    END;                                            00709800
END;                                              00709900
IF IDCOUNTER > 0                                  00710000
THEN                                              00710100
BEGIN                                            00710200
    $ENTER(IDSTACK,CURRENTPROCEDUREID,              00710300
            FORMALPARMPTR: PTSTRUCTURETAB-1,        00710400
            NUMBROFPARMS: IDCOUNTER)                00710500
END;                                              00710600
$EXEC(SCANSUBSTRUCTURE);                          00710700
END;                                              00710800
$ACTION(ENDOFACCESSPARAMETERSCAN);                00710900
COMMENT THIS ACTION CLEANS UP AFTER THE PARAMETERS IN AN ACCESS 00711000
DECLARATION HAVE BEEN SCANNED;                    00711100

```

```

$EXEC(ENDOFACTUALPARAMETERSCAN);
$ACTION(ENDOFACCESSPROCEDUREBODY);
    COMMENT THIS TERMINATES AN ACCESS PROCEDURE BODY;
$EXEC(ENDOFPROCEDUREBODY);
$ACTION(ENDOFACTUALPARAMETERSCAN);
    COMMENT THIS ACTION POPS THE ACTUAL PARAMETER SCANNING
    VARIABLES OFF OF ACTUALPARMSTACK AND TERMINATES AN ACTUAL
    PARAMETER SCAN;
    DROP
    IF CURRENTACTUALPARAMETERCOUNT < CURRENTNUMBEROFFORMALPARAMETERS
    THEN
    BEGIN
        ERRMS "TOO FEW ACTUAL PARAMETERS HAVE BEEN SUPPLIED." ;
        WRITEMESOUT
    END;
    $POP(ACTUALPARMSTACK,CURRENTACTUALPARAMETERCOUNT);
    $POP(ACTUALPARMSTACK,CURRENTNUMBEROFFORMALPARAMETERS);
    $POP(ACTUALPARMSTACK,CURRENTFORMALPARAMETERLISTPTR);
    END;
$ACTION(ENDOFBLOCK);
    COMMENT THIS ACTION POPS ALL OF THE VARIABLES OFF OF IDSTACK
    AND STRUCTURETAB THAT WERE DECLARED IN THE LAST BLOCK;
    DROP
    $POP(IDCOUNTERSTACK;IDCOUNTER);
    FOR I + 1 STEP 1 UNTIL IDCOUNTER DO
    BEGIN
        $ENTER(RIGTAB,$OPD(IDSTACK,$TOP,RIGTABPTR),
            SEM: $OPD(IDSTACK,$TOP,OLDSEMFIELD));
        IF $OPD(IDSTACK,$TOP,PROCEDURE)=1
            OR $OPD(IDSTACK,$TOP,TYPE)=STRUCT
            THEN PTSTRUCTURETAB + $OPD(IDSTACK,$TOP,STRUCTPTRBEFORE) ;
        $POP(IDSTACK);
    END;
    IDCOUNTER + 0;
    END;
$ACTION(ENDOFDATADECLARATION);
    COMMENT THIS ACTION CLEANS UP ANY OWN, NODE, ETC. FLAGS THAT
    MAY HAVE BEEN LEFT ON WHILE SCANNING A DATA DECLARATION ;
    CURRENTDECLTYPE + CURRENTDECLOWN + CURRENTDECLNODE +
    CURRENTDECLMACRO + CURRENTACCESSID + 0;
$ACTION(ENDOFDECLIST);
    COMMENT RESET ISDEFINED AND ISREINGDEFINED BITS IN BIGTAB FOR
    EACH ID ENCOUNTERED. THEN PUSH THE ID COUNT ;
    DROP
    $PUSH(IDCOUNTERSTACK;IDCOUNTER);
    FOR I + 1 STEP 1 UNTIL IDCOUNTER DO
    BEGIN
        $ENTER(RIGTAB,$OPD(IDSTACK,$TOP+1-1,BIGTABPTR),
            ISDEFINED:1,
            ISREINGDEFINED:0);
    END;
    IDCOUNTER + 0;
    END;
$ACTION(ENDOFENTIRESTRUCTUREDECLARATION);
    COMMENT THIS ACTION CLEANS UP AFTER AN ENTIRE STRUCTURE
    DECLARATION HAS BEEN SCANNED;
    STRUCTUREISOWN + FALSE;
$ACTION(ENDOFFORMALS);
    COMMENT THIS ACTION RESETS ALL FORMAL PARAMETER SCANNING
    SWITCHES;
    $POP(FORMALSWITCHSTACK;SCANNINGFORMALPARAMETERDECLARATIONS);

```

```

00711200
00711300
00711400
00711410
00711500
00711600
00711700
00711800
00711900
00712000
00712100
00712200
00712300
00712400
00712500
00712600
00712700
00712800
00712900
00713000
00713100
00713200
00713300
00713400
00713500
00713600
00713700
00713800
00713900
00714000
00714100
00714200
00714300
00714400
00714500
00714600
00714700
00714800
00714900
00715000
00715100
00715200
00715300
00715400
00715500
00715600
00715700
00715800
00715900
00716000
00716100
00716200
00716300
00716400
00716500
00716600
00716700
00716800
00716900
00717000
00717100

```

```

$ACTION(ENDDPROCEDUREBODY):
    COMMENT THIS ACTION POPS THE FORMAL PARAMETER DECLARATIONS AND
    RESETS IDSTACK TO ITS CONDITION PRIOR TO CALLING THIS PROCEDURE;
    DROP
    $POP(PROCEDUREIDSTACK,CURRENTPROCEDUREID);
    $EXEC(ENDDSTRUCTUREDECLARATION);
    $EXEC(ENDDSTRUCTUREDECLARATION);
    END;
$ACTION(ENDDPROCEDUREHEADING):
    COMMENT THIS ACTION TESTS THAT ALL FORMAL PARAMETERS WERE
    DECLARED AND DOES AN END OF BLOCK THING BEFORE ENTERING THE
    PROCEDURE BODY. IT ALSO CHECKS THAT FORWARD DECLARED PROCEDURE
    HEADINGS ARE COMPATIBLE WITH THE FINAL DECLARATION;
    DROP
    FOR I ← 1 STEP 1 UNTIL IDCOUNTER DO
    BEGIN
        $PUSH(STRUCTURETAB,$ALLOF(IDSTACK,$TOP+1-1));
        IF $OPD(IDSTACK,$TOP,HASBEENSPEC) = 0
        THEN
            BEGIN
                ERRMS "FORMAL PARAMETER NOT DECLARED IN PROCEDURE ",
                    "HEADING.";
                WRITEMESOUT;
            END;
        END;
    IF IDCOUNTER > 0
    THEN
        BEGIN
            $ENTER(IDSTACK,CURRENTPROCEDUREID,
                FORMALPARMPTR : PTSTRUCTURETAB-1,
                NUMBROFFPARMS : IDCOUNTER)
        END;
    IF $OPD(BIGTAB,$OPD(IDSTACK,CURRENTPROCEDUREID,RIGTARPTR),
        FORWARDDEFINED) = 1
    COMMENT NOTE THAT FORWARDDEFINED IN IDSTACK IS FROM THE
    OLDSEMFIELD OR PREVIOUS DEFINITION OF THIS IDENTIFIER;
    THEN
        BEGIN COMMENT CHECK THIS DUDE OUT FOR COMPATABILITY WITH HIS
            FORWARD DEFINITION;
            IF $OPD(IDSTACK,CURRENTPROCEDUREID,NUMBROFFPARMS) ≠ IDCOUNTER
            THEN NOFORWARDMATCH
            ELSE
                BEGIN
                    LABEL LOOP;
                    I ← PTSTRUCTURETAB;
                    J ← $OPD(IDSTACK,CURRENTPROCEDUREID,FORMALPARMPTR)+1;
                    FOR K ← 1 STEP 1 UNTIL IDCOUNTER DO
                    BEGIN
                        IF NOT EQUALSTRUCTURETAB(I-K,J-K)
                        THEN
                            BEGIN
                                NOFORWARDMATCH;
                                LEAVE LOOP
                            END
                        END; LOOP;
                    END;
                END;
            END;
        $EXEC(SCANSUBSTRUCTURE);
        END;
$ACTION(ENDDSTRUCTUREDECLARATION):
    COMMENT THIS ACTION TERMINATES THE SCAN OF A STRUCTURE

```

00717200
 00717300
 00717400
 00717500
 00717600
 00717700
 00717800
 00717900
 00718000
 00718100
 00718200
 00718300
 00718400
 00718500
 00718600
 00718700
 00718800
 00718900
 00719000
 00719100
 00719200
 00719300
 00719400
 00719500
 00719600
 00719700
 00719800
 00719900
 00720000
 00720100
 00720200
 00720300
 00720400
 00720500
 00720600
 00720700
 00720800
 00720900
 00721000
 00721100
 00721200
 00721300
 00721400
 00721500
 00721600
 00721700
 00721800
 00721900
 00722000
 00722100
 00722200
 00722300
 00722400
 00722500
 00722600
 00722700
 00722800
 00722900
 00723000
 00723100
 00723200

DECLARATION LIST. THE IDLISTCOUNTER IS POPPED FROM STRUCTUREPARM- 00723300
 STACK AND IDENTICAL ENTRIES ARE MADE IN STRUCTURETAB FOR EACH ID IN 00723400
 THE ID LIST ASSOCIATED WITH THIS STRUCTURE. THE RESULT IS ONE ID 00723500
 ENTRY FOR THE STRUCTURE IN THE ID STACK AND A POINTER TO AN ARRAY 00723600
 OF ID ENTRIES IN STRUCTURETAB. THE ID ENTRIES IN STRUCTURETAB ARE 00723700
 THE IDS IN THE SUB-STRUCTURE IN THE ORDER IN WHICH THEY OCCURRED 00723800
 IN THE SUB-STRUCTURE DECLARATION. SHOULD ANY OF THESE IDS IN 00723900
 STRUCTURETAB REPRESENT SUBSTRUCTURES WITHIN THE CURRENTLY DEFINED 00724000
 STRUCTURE, THEY WILL POINT TO ANOTHER SUB-STRUCTURE ID ARRAY IN 00724100
 STRUCTURETAB. BIGTAB ENTRIES POINTING TO THE SUB-STRUCTURE IDS 00724200
 ARE POPPED AND ALL IDS IN THE CURRENT LIST OF DECLARATIONS ARE SET 00724300
 TO BEING DEFINED AGAIN; 00724400
 00724500

E.G. STRUCTURE A(INTEGER A,B) STRUCTURE X(POINTER LEFT,RIGHT); 00724600
 COMMENT STRING NAME) 00724700
 STRUCTURE ENTRIES: 00724800
 00724900

BIGTAB	IDSTACK (BIGTABPTR)	STRUCTURETAB	
+	-----+		00725000
:		:	00725100
:	(BIGTABPTR)	:	00725200
:	<-----+	:	00725300
:	:	:	00725400
V (IDSTACKPTR)	:	(STRUCTUREPTR)	00725500
A:----->	A:----->	A:(INTEGER)	00725600
	(STRUCT-4 ENTRIES)		00725700
	(BIGTABPTR)		00725800
+	-----+		00725900
:		:	00726000
V (IDSTACKPTR)		:	00726100
B:----->	(NULL)	B:(INTEGER)	00726200
			00726300
	(BIGTABPTR)		00726400
+	-----+		00726500
:		:	00726600
V (IDSTACKPTR)		:	00726700
X:----->	(NULL)	X:(STRUCT-2 ENTRIES)---	00726800
			00726900
	(BIGTABPTR)		00727000
+	-----+		00727100
:		:	00727200
:		:	00727300
+	(IDSTACKPTR)		00727400
+-> LEFT:----->	(NULL)	NAME:(STRING)	00727500
:		:	00727600
:	(BIGTABPTR)	:	00727700
+	-----+		00727800
:		:	00727900
:	(IDSTACKPTR)	:	00728000
+-> RIGHT:----->	(NULL)	+(STRUCTUREPTR)-----+	00728100
:		:	00728200
:	(BIGTABPTR)	:	00728300
+	-----+	:	00728400
:		:	00728500
:	(IDSTACKPTR)	:	00728600
-----> NAME:----->	(NULL)	-----> LEFT:(POINTER)	00728700
			00728800
			00728900
			00729000
			00729100
			00729200
			00729300

```

                                RIGHT:(POINTER)                                00729400
                                                                00729500
                                                                00729600
WHEN THIS ACTION IS COMPLETE, THE IDSTACK WILL CONTAIN ONE NEW ID 00729700
ENTRY FOR EACH ID IN THE ID PART. THESE IDS WILL BE ON THE TOP OF 00729800
THE IDSTACK AND ALL IDS CONTAINED WITHIN THE SUB-STRUCTURE WILL 00729900
HAVE BEEN REMOVED FROM THE STACK. A COMPLETE STRUCTURE DESCRIPTION 00730000
WILL EXIST IN STRUCTURE TAB FOR EACH ID IN THE ID PART; 00730100
DROP 00730200
$POP(STRUCTUREPARAMSTACK:CURRENTDECLTYPE); 00730300
$POP(STRUCTUREPARAMSTACK:CURRENTDECLMACRO); 00730400
$POP(STRUCTUREPARAMSTACK:CURRENTDECLNODE); 00730500
$POP(STRUCTUREPARAMSTACK:IDLISTCOUNTER); 00730600
$POP(STRUCTUREPARAMSTACK:CURRENTDECLDOWN); 00730700
$POP(FORMALSWITCHSTACK:SCANNINGFORMALPARAMETERDECLARATIONS); 00730800
COMMENT MAKE ENTRIES IN STRUCTURETAB; 00730900
FOR I ← 1 STEP 1 UNTIL IDLISTCOUNTER DO 00731000
BEGIN 00731100
    FOR J ← 1 STEP 1 UNTIL IDCOUNTER DO 00731200
    BEGIN 00731300
        $PUSH(STRUCTURETAB,$ALLOF(IDSTACK,$TOP+1-J)); 00731400
    END; 00731500
    $ENTER(IDSTACK,$TOP+1-IDCOUNTER-1, 00731600
            STRUCTUREPTR: PTSTRUCTURETAB-1, 00731700
            NUMBRSUBSTRUCT:IDCOUNTER); 00731800
END; 00731900
COMMENT PUT SEM FIELDS BACK INTO BIGTAB; 00732000
FOR I ← 1 STEP 1 UNTIL IDCOUNTER DO 00732100
BEGIN 00732200
    $ENTER(BIGTAB,$OPD(IDSTACK,$TOP,BIGTABPTR), 00732300
            SEM: $OPD(IDSTACK,$TOP,OLDSEMFIELD)); 00732400
    $POP(IDSTACK); 00732500
    $DECR(CURRENTMAXIMUMDISPLACEMENT); 00732600
END; 00732700
COMMENT RESET ISBEINGDEFINED BITS IN BIGTAB FOR ALL CURRENT 00732800
BLOCK IDS; 00732900
$POP(IDCOUNTERSTACK:IDCOUNTER); 00733000
FOR I ← 1 STEP 1 UNTIL IDCOUNTER DO 00733100
BEGIN 00733200
    $ENTER(BIGTAB,$OPD(IDSTACK,$TOP+1-1,BIGTABPTR), 00733300
            ISDEFINED:0, 00733400
            ISBEINGDEFINED:1); 00733500
END; 00733600
END; 00733700
$ACTION(ENTERACCESSPROCEDUREID); 00733800
COMMENT THIS ACTION MARKS A PROCEDURE TO BE AN ACCESS PROCEDURE 00733900
FOR LATER COLLECTION INTO STRUCTURETAB. IT ALSO MAKES A FORMAL 00734000
PARAMETER ENTRY OF THE FIRST PARAMETER. THIS PARAMETER IS THE 00734100
QUEUE TYPE. DOING THIS FORCES LATER SYNTAX TO REQUIRE THAT THE 00734200
QUEUE NAME APPEAR IN THE FIRST PARAMETER SLOT IN THE PROCEDURE 00734300
STATEMENT OR FUNCTION CALL; 00734400
DROP 00734500
$ENTER(IDSTACK,$TOP,ACCESSPROCEDURE:1); 00734600
$PUSH(IDSTACK,$ALL 0: 0, 00734700
        $ALL 1: 0, 00734800
        $ALL 2: 0, 00734900
        $ALL 3: 0, 00735000
        BIGTABPTR: (I+$OPD(IDSTACK, 00735100
                        $OPD(PROCEDUREIDSTACK,$TOP), 00735200
                        BIGTABPTR)), 00735300
        OLDSEMFIELD: $OPD(BIGTAB,I,SEM), 00735400

```

```

        FORMALPARAM: 1, 00735500
        HASRENSPEC: 1, 00735600
        TYPE: ACCESSVAR, 00735700
        ACCESTYPE: CURRENTMAXACCESSNUMBER, 00735800
        DUMMYPARM: 1)) 00735900
$INCR(IDCOUNTER); 00736000
END; 00736100
$ACTION(ENTERIDINDECLSTACK); 00736200
DROP 00736300
IF SCANNINGFORMALPARAMETERDECLARATIONS 00736400
THEN 00736500
BEGIN 00736600
    IF NOT BOOLEAN($OPD(BIGTAB,$OPD(MSTACK,$TOP,ENTRY),
        ISBEINGDEFINED)) 00736700
    THEN 00736800
    BEGIN 00736900
        ERRMS "THIS ID IS NOT A FORMAL PARAMETER."; 00737000
        WRITEMESOUT 00737100
    END 00737200
    ELSE 00737300
    BEGIN 00737400
        $ENTER(IDSTACK,I+$OPD(BIGTAB, 00737500
            $OPD(MSTACK,$TOP,ENTRY), 00737600
            IDSTACKPTR), 00737700
            TYPE: CURRENTDECLTYPE, 00737800
            NODE: CURRENTDECLNODE, 00737900
            MACRO: CURRENTDECLMACRO, 00738000
            LEVEL: CURRENTLEVEL, 00738100
            DISPLACEMENT: CURRENTMAXIMUMDISPLACEMENT, 00738200
            HASRENSPEC: 1)); 00738300
        $INCR(CURRENTMAXIMUMDISPLACEMENT); 00738400
        IF CURRENTACCESSID # 0 AND CURRENTDECLTYPE = ACCESSVAR 00738500
        THEN 00738600
        BEGIN 00738700
            $ENTER(IDSTACK,I, 00738800
                ACCESTYPE: 00738900
                $OPD(IDSTACK,CURRENTACCESSID,ACCESTYPE), 00739000
                STORETYPE: 00739100
                $OPD(IDSTACK,CURRENTACCESSID,STORETYPE), 00739200
                FETCHTYPE: 00739300
                $OPD(IDSTACK,CURRENTACCESSID,FETCHTYPE), 00739400
                FETCHPROCEDURE: 00739500
                $OPD(IDSTACK,CURRENTACCESSID,FETCHPROCEDURE), 00739600
                STOREPROCEDURE: 00739700
                $OPD(IDSTACK,CURRENTACCESSID,STOREPROCEDURE), 00739800
                FETCHAUXPTR: 00739900
                $OPD(IDSTACK,CURRENTACCESSID,FETCHAUXPTR), 00740000
                STOREAUXPTR: 00740100
                $OPD(IDSTACK,CURRENTACCESSID,STOREAUXPTR)) 00740200
            END; 00740300
            $INCR(IDLISTCOUNTER); 00740400
        END 00740500
    END 00740600
ELSE 00740700
IF BOOLEAN($OPD(BIGTAB,$OPD(MSTACK,$TOP,ENTRY),ISBEINGDEFINED)) 00740800
THEN 00740900
BEGIN COMMENT IF THIS IDENTIFIER HAS ALREADY BEEN DECLARED IN 00741000
    THIS BLOCK, THEN PRINT A MESSAGE TO THAT EFFECT ; 00741100
    ERRMS "ID USED TWICE IN SAME BLOCK - ID IGNORED"; 00741200
    WRITEMESOUT 00741300
END 00741400
END 00741500

```

```

ELSE
BEGIN
    COMMENT PUSH A NEW ID INTO THE ID STACK. PUT THE OLD
    BIGTAB SEMANTICS FIELD INTO THE NEW ID AND SET DEFAULTS FOR
    THIS DATA TYPE. THEN SET THE BIGTAB POINTER IN THE NEW ID
    STACK TO POINT BACK TO THE BIGTAB ENTRY, AND, FINALLY, ADD ONE
    TO THE IDCOUNTER TO INDICATE HOW MANY IDS HAVE BEEN PROCESSED
    INTO THE ID STACK AT THIS BLOCK LEVEL ;
    $PUSH(IDSTACK, $ALL 0: 0,
           $ALL 1: 0,
           $ALL 2: 0,
           $ALL 3: 0,
           TYPE: CURRENTDFCLTYPE,
           BIGTABPTR: $OPD(MSTACK,$TOP,ENTRY),
           LEVEL: CURRENTLEVEL,
           DISPLACEMENT: CURRENTMAXIMUMDISPLACEMENT,
           OWN: CURRENTDFCLOWN,
           NODE: CURRENTDFCLNODE,
           OLDSFMFIELD: $OPD(BIGTAB,
                             $OPD(MSTACK,$TOP,ENTRY),
                             SEM),
           MACRO: CURRENTDFCLMACRO);
    $ENTER(BIGTAB, $OPD(MSTACK,$TOP,ENTRY),
           SEM: 0,
           ISPREINGDEFINED: 1,
           IDSTACKPTR: PTIDSTACK-1);
    IF PRINTOBJECT
    THEN
    BEGIN
        BLANKMESOUT;
        REPLACE PMESOUT BY "(",CURRENTLEVEL FOR 2 DIGITS,
        " ",CURRENTMAXIMUMDISPLACEMENT FOR 4 DIGITS,
        ") = ",POINTER($OPD(BIGTAB,$OPD(MSTACK,$TOP,
        ENTRY)+1,$ALL)) FOR
        $OPD(BIGTAB,$OPD(MSTACK,$TOP,ENTRY),IDLENGTH);
        WRITEMESOUT;
    END;
    $INCR(CURRENTMAXIMUMDISPLACEMENT);
    IF CURRENTACCESSID # 0 AND CURRENTDECLTYPE = ACCESSVAR
    THEN
    BEGIN
        $ENTER(IDSTACK,I,
              ACCESTYPE:
              $OPD(IDSTACK,CURRENTACCESSID,ACCESTYPE),
              STORETYPE:
              $OPD(IDSTACK,CURRENTACCESSID,STORETYPE),
              FETCHTYPE:
              $OPD(IDSTACK,CURRENTACCESSID,FETCHTYPE),
              FETCHPROCEDURE:
              $OPD(IDSTACK,CURRENTACCESSID,FETCHPROCEDURE),
              STOREPROCEDURE:
              $OPD(IDSTACK,CURRENTACCESSID,STOREPROCEDURE),
              FETCHAUXPTR:
              $OPD(IDSTACK,CURRENTACCESSID,FETCHAUXPTR),
              STOREAUXPTR:
              $OPD(IDSTACK,CURRENTACCESSID,STOREAUXPTR));
    END;
    $INCR(IDLISTCOUNTER);
    $INCR(IDCOUNTER);
END;
END;

```

```

ACTION(CENTERNUMBEROFSUBSCRIPTS):                                00747700
DROP                                                                00747800
FOR I ← 1 STEP 1 UNTIL IDLISTCOUNTER DO                            00747900
BEGIN                                                                00748000
    $ENTER(IDSTACK,$TOP+1-1,                                        00748100
            SUBSCRIPTED: 1,                                         00748200
            NUMBROFSUBS: CURRENTNUMBROFSUBS);                      00748300
END;                                                                00748400
CURRENTNUMBROFSUBS ← 0;                                           00748500
END;                                                                00748600
ACTION(EVENTDECLARATION):                                          00748700
COMMENT THIS ACTION SETS CURRENTDECLTYPE = EVNT TO INDICATE      00748800
THAT AN EVENT DECLARATION IS BEING SCANNED;                      00748900
CURRENTDECLTYPE ← EVNT;                                           00749000
ACTION(FILEDECLARATION):                                           00749100
COMMENT THIS ACTION SETS CURRENT DECLTYPE = OSLFILE;             00749200
CURRENTDECLTYPE ← OSLFILE;                                         00749300
ACTION(FORMALPARAMETER):                                           00749400
COMMENT THIS ACTION ENTERS A FORMAL PARAMETER INTO IDSTACK.      00749500
THE FORMAL PARAMETER BIT IS TURNED ON. ALSO, THE IS BEING DEFINED 00749600
BIT IS TURNED ON. IF THE IS BEING DEFINED BIT IS ALREADY ON THEN 00749700
AN ERROR MESSAGE IS PRINTED;                                       00749800
DROP                                                                00749900
IF BOOLFAN($OPD(RIGTAB,$OPD(MSTACK,$TOP,ENTRY),ISBEINGDEFINED)) 00750000
THEN                                                                00750100
BEGIN                                                                00750200
    ERRMS "ID USED TWICE IN THE SAME FORMAL PARAMETER LIST - ",   00750300
    "SECOND ID IS IGNORED.";                                       00750400
    WRITEMFSOUT                                                    00750500
END                                                                00750600
ELSE                                                                00750700
BEGIN                                                                00750800
    $PUSH(IDSTACK, $ALL 0: 0,                                       00750900
           $ALL 1: 0,                                              00751000
           $ALL 2: 0,                                              00751100
           $ALL 3: 0,                                              00751200
           RIGTABPTR: $OPD(MSTACK,$TOP,ENTRY),                     00751300
           OLDSEMFIELD: $OPD(RIGTAB,                                00751400
                             $OPD(MSTACK,$TOP,ENTRY),             00751500
                             SEM),                                 00751600
           FORMALPARM: 1);                                          00751700
    $ENTER(RIGTAB, $OPD(MSTACK,$TOP,ENTRY),                        00751800
           SEM: 0,                                                  00751900
           ISREINGDEFINED:1,                                       00752000
           IDSTACKPTR: PTIDSTACK-1);                               00752100
    $INCR(IDCOUNTER);                                              00752200
END;                                                                00752300
END;                                                                00752400
ACTION(FUNCTIONISBOOLEAN): SEMANTICTEST ← FUNCTIONTEST(BOOL);    00752500
ACTION(FUNCTIONISINTEGER): SEMANTICTEST ← FUNCTIONTEST(INT);      00752600
ACTION(FUNCTIONISPATTERN): SEMANTICTEST ← FUNCTIONTEST(PATT);     00752700
ACTION(FUNCTIONISPOINTER): SEMANTICTEST ← FUNCTIONTEST(PTR);      00752800
ACTION(FUNCTIONISSTRING): SEMANTICTEST ← FUNCTIONTEST(STR);       00752900
ACTION(FUNCTIONISSTRUCTURE): SEMANTICTEST ← FUNCTIONTEST(STRUCT); 00753000
ACTION(GETNEXTPARM):                                              00753100
COMMENT THIS ACTION UPDATES THE ACTUAL PARAMETER COUNT;          00753200
$INCR(CURRENTACTUALPARAMETERCOUNT);                              00753300
ACTION(IDISACCESS):                                              00753400
COMMENT THIS ACTION SETS SEMANTICTEST TRUE IF THE IDENTIFIER     00753500
JUST SCANNED IS AN ACCESS VARIABLE;                                00753600
SEMANTICTEST ← $OPD(IDSTACK,$OPD(RIGTAB,$OPD(MSTACK,$TOP,ENTRY), 00753700

```

```

                                IDSTACKPTR),TYPE) = ACCESSVAR; 00753800
                                ); 00753900
$ACTION( IDISPROCEDURE
    COMMENT THIS ACTION SETS SEMANTICTEST TRUE IF THE ID ON THE TOP 00754000
    OF MSTACK IS A PROCEDURE ID ; 00754100
    DROP 00754200
    IF CURRENTSTRUCTUREPTR = 0 00754300
    THEN 00754400
    BEGIN 00754500
        IF $OPD(BIGTAB,$OPD(MSTACK,$TOP,ENTRY),IDSTACKPTR) = 0 00754600
        THEN SEMANTICTEST ← FALSE 00754700
        ELSE SEMANTICTEST ← BOOLEAN($OPD(IDSTACK, 00754800
                                $OPD(BIGTAB, 00754900
                                $OPD(MSTACK,$TOP,ENTRY), 00755000
                                IDSTACKPTR), 00755100
                                PROCEDURE)); 00755200

        IF SEMANTICTEST 00755300
        THEN 00755400
        BEGIN 00755500
            $ENTER(MSTACK,$TOP, 00755600
                PROCEDUREPTR: $OPD(BIGTAB, 00755700
                    $OPD(MSTACK,$TOP,ENTRY), 00755800
                    IDSTACKPTR), 00755900
                PROCEDUREPTRISINTOIDSTACK: 1); 00756000

        END 00756100
    END 00756200
    ELSE 00756300
    BEGIN 00756400
        INTEGER I; 00756500
        I ← FINDSUBSTRUCTUREID($OPD(MSTACK,$TOP,ENTRY)); 00756600
        SEMANTICTEST ← IF I = 0 00756700
        THEN FALSE 00756800
        ELSE BOOLEAN($OPD(STRUCTURETAB,I,PROCEDURE)); 00756900

        IF SEMANTICTEST 00757000
        THEN 00757100
        BEGIN 00757200
            CURRENTIDPTR ← I; 00757300
            $ENTER(MSTACK,$TOP, 00757400
                PROCEDUREPTR: 1, 00757500
                PROCEDUREPTRISINTOIDSTACK: 0) 00757600

        END 00757700
    END 00757800
    END; 00757900
$ACTION( INCRSUBSCRIPT ); 00758000
    COMMENT THIS ACTION ADDS ONE TO THE SUBSCRIPT COUNTER; 00758100
    DROP 00758200
    $INCR(CURRENTNUMBEROFSUBS); 00758300
    OPERATOR("MAKE-INDEX"); 00758400
    END; 00758500
$ACTION( INTEGERDECLARATION); 00758600
    COMMENT THIS ACTION SETS CURRENTDECLTYPE = INT TO INDICATE THAT 00758700
    WE ARE NOW PROCESSING AN INTEGER DECLARATION ; 00758800
    CURRENTDECLTYPE ← INT; 00758900
$ACTION( INTEGERPROCEDURE); 00759000
    DROP 00759100
    SEMANTICTEST ← (IF CURRENTSTRUCTUREPTR = 0 00759200
        THEN $OPD(IDSTACK,CURRENTPROCEDUREID,TYPE) 00759300
        ELSE $OPD(STRUCTURETAB,CURRENTIDPTR,TYPE)) 00759400
        = INT ; 00759500

    END; 00759600
$ACTION( INTEGERSTORE); 00759700
    DROP 00759800

```

```

OPERATOR("STORE-INTEGERS");                                00759900
END;                                                         00760000
ACTION(ISIDANACCESSTYPE);                                   00760100
    COMMENT THIS ACTION SETS SEMANTICTEST = TRUE IF THE ID JUST 00760200
    SCANNED IS AN ALLOWABLE ACCESS TYPE;                     00760300
    SEMANTICTEST + $OPD(IDSTACK,$OPD(RIGTAB,$OPD(MSTACK,$TOP,ENTRY), 00760400
        IDSTACKPTR),TYPE) = ACCESS ;                          00760500
ACTION( LEAVELABEL );                                       00760600
    COMMENT THIS ACTION CHECKS THAT THE LABEL IN A LEAVE STATEMENT 00760700
    IS LEGITIMATE AT THIS TIME;                               00760800
    DROP                                                      00760900
    I + $OPD(MSTACK,$TOP,ENTRY);                              00761000
    J + $OPD(RIGTAB,I,IDSTACKPTR);                            00761100
    IF $OPD(IDSTACK,J,TYPE) ≠ LARL                             00761200
    THEN                                                       00761300
    BEGIN                                                      00761400
        ERRMS "THIS IS NOT A LABEL.";                          00761500
        WRITEMESOUT;                                           00761600
    END                                                         00761700
    ELSE IF J < CURRENTPROCEDUREID                            00761800
    THEN                                                       00761900
    BEGIN                                                      00762000
        ERRMS "THIS LABEL IS OUTSIDE THE RANGE OF THIS PROCEDURE."; 00762100
        WRITEMESOUT;                                           00762200
    END                                                         00762300
    ELSE                                                       00762400
    BEGIN                                                      00762500
        $PUSH(LEAVESTACK,$ALL:0,ADDRESS:CURRENTADDRESS);      00762600
        OPERATOR("*** (BRANCH INSTRUCTION WILL BE HERE) ***"); 00762700
    END;                                                       00762800
    END;                                                       00762900
ACTION(MACRODEFINITION);                                    00763000
    COMMENT THIS ACTION SETS CURRENTDECLMACRO=1 SO IT MAY BE USED 00763100
    TO INDICATE MACRO VARIABLES AND PROCEDURES;              00763200
    CURRENTDECLMACRO + 1;                                       00763300
ACTION(MAKESURENOPARAMETERSAREREQUIRED);                   00763400
    COMMENT THIS ACTION MAKES SURE THAT NO PARAMETERS MUST BE    00763500
    SUPPLIED TO AN ACCESS DECLARATION;                         00763600
    DROP                                                        00763700
    IF $OPD(IDSTACK,CURRENTACCESSID,NUMPROFPARMS) ≠ 0          00763800
    THEN                                                       00763900
    BEGIN                                                      00764000
        ERRMS "PARAMETERS MUST BE SUPPLIED WITH ",            00764100
        POINTER($OPD(RIGTAB,(I+$OPD(IDSTACK,CURRENTACCESSID, 00764200
            RIGTARPTR))+1,$ALL))                               00764300
        FOR $OPD(RIGTAB,I,IDLENGTH),                           00764400
        " ACCESS DECLARATIONS.";                               00764500
        WRITEMESOUT                                             00764600
    END;                                                       00764700
    END;                                                       00764800
ACTION(MOVEOSL2SEMFIELDFROM1TO4);                           00764900
    COMMENT THIS ACTION MOVES THE OSL2SEMFIELD IN MSTACK FROM    00765000
    TOP-1 TO TOP-4;                                             00765100
    $OPD(MSTACK,$TOP-4,OSL2SEMFIELD)+$OPD(MSTACK,$TOP-1,OSL2SEMFIELD); 00765200
ACTION(MOVEOSL2SEMFIELDFROM1TO6);                           00765300
    COMMENT THIS ACTION MOVES THE OSL2SEMFIELD IN MSTACK FROM    00765400
    TOP-1 TO TOP-6;                                             00765500
    $OPD(MSTACK,$TOP-6,OSL2SEMFIELD)+$OPD(MSTACK,$TOP-1,OSL2SEMFIELD); 00765600
ACTION(MUSTBEUNSUBSCRIPTED);                                00765700
    COMMENT THIS ACTION PRINTS AN ERROR MESSAGE IF A SUBSCRIPTED 00765800
    VARIABLE APPEARS WITHOUT ITS SUBSCRIPT;                    00765900

```

```

DROP                                                    00766000
IF( IF CURRENTSTRUCTUREPTR=0 THEN $OPD(IDSTACK,        00766100
                                $OPD(BIGTAB,           00766200
                                $OPD(MSTACK,$TOP,ENTRY), 00766300
                                IDSTACKPTR),           00766400
                                SUBSCRIPTED)           00766500
ELSE $OPD(STRUCTURETAB,    00766600
          CURRENTIDPTR,    00766700
          SUBSCRIPTED))=1  00766800

THEN                                                    00766900
BEGIN                                                  00767000
    ERRMS "SUBSCRIPT REQUIRED." 00767100
    WRITEMESOUT                00767200
END;                                                    00767300
END;                                                    00767400
$ACTION(NODEDECLARATION);                             00800100
    COMMENT THIS ACTION SETS CURRENTDECLNODE = 1 TO INDICATE THAT 00800200
    ALL IDENTIFIERS IN THIS DECLARATION ARE OF TYPE NODE ;      00800300
    CURRENTDECLNODE ← 1;                                       00800400
$ACTION( OPENLABEL                                     00800500
                                );
    COMMENT THIS ACTION ENTERS A LABEL IN IDSTACK AS IF IT WERE A 00800600
    DECLARATION;                                              00800700
DROP                                                        00800800
$PUSH(IDSTACK, $ALL 0: 0, 00800900
        $ALL 1: 0, 00801000
        $ALL 2: 0, 00801100
        $ALL 3: 0, 00801200
        TYPE: LARL, 00801300
        LABELSEGMENT: CURRENTSEGMENT, 00801400
        LABELSTARTADDRESS: CURRENTADDRESS, 00801500
        BIGTABPTR:$OPD(MSTACK,$TOP,ENTRY), 00801600
        OLDSEMFIELD:$OPD(BIGTAB,$OPD(MSTACK,$TOP,ENTRY), 00801700
                          SEM)); 00801800
$ENTER(BIGTAB,$OPD(MSTACK,$TOP,ENTRY), 00801900
        SEM: 0, 00802000
        ISDEFINED: 1, 00802100
        IDSTACKPTR: PTIDSTACK-1); 00802200
    $PUSH(IDCOUNTERSTACK,1); 00802300
    OPERFIRSTPART,"LABEL ",PRINTID;OPERLASTPART; 00802400
    $DECR(CURRENTADDRESS); 00802500
    $PUSH(LFAVESTACK,$ALL 0,MARK:1); 00802600
END; 00802700
$ACTION(OWNDECLARATION); 00802800
    COMMENT THIS ACTION RECOGNIZES THE RESERVED WORD OWN AND SETS 00802900
    CURRENTDECLOWN = 1 TO INDICATE THAT ALL IDENTIFIERS IN THIS 00803000
    DECLARATION ARE OF TYPE OWN; 00803100
    CURRENTDECLOWN ← 1; 00803200
$ACTION(PATTERNDECLARATION); 00803300
    COMMENT THIS ACTION SETS CURRENTDECLTYPE = PATT TO INDICATE 00803400
    THAT ALL FOLLOWING IDS IN THIS DECLARATION ARE OF TYPE PATTERN; 00803500
    CURRENTDECLTYPE ← PATT; 00803600
$ACTION(PATTERNPROCEDURE); 00803700
DROP 00803800
SEMANTICTEST ← (IF CURRENTSTRUCTUREPTR = 0 00803900
                THEN $OPD(IDSTACK,CURRENTPROCEDUREID,TYPE) 00804000
                ELSE $OPD(STRUCTURETAB,CURRENTIDPTR,TYPE)) 00804100
                = PATT ; 00804200
END; 00804300
$ACTION(PATTERNSTORE); 00804400
DROP 00804500
OPERATOR("STORE-PATTERN"); 00804600

```

```

END;
ACTION(POINTERDECLARATION):
    COMMENT THIS ACTION SETS CURRENTDECLTYPE = PTR TO INDICATE THAT
    A POINTER DECLARATION IS BEING SCANNED;
    CURRENTDECLTYPE ← PTR;
ACTION(POINTERPROCEDURE):
    DROP
    SEMANTICTEST ← (IF CURRENTSTRUCTUREPTR = 0
        THEN $OPD(IDSTACK,CURRENTPROCEDUREID,TYPE)
        ELSE $OPD(STRUCTURETAB,CURRENTIDPTR,TYPE))
        = PTR ;
END;
ACTION(POINTERSTORE):
    DROP
    OPERATOR("STORE-POINTER");
END;
ACTION(PROCEDUREIDENTIFIER):
    COMMENT THIS ACTION ENTERS NON-STRUCTURE PROCEDURE IDS IN THE
    IDSTACK AND UPDATES STRUCTURE IDS TO BE CURRENT;
    DROP
    IF SCANNINGFORMALPARAMETERDECLARATIONS
    THEN
    BEGIN
        IF NOT BOOLEAN($OPD(RIGHTAR,$OPD(MSTACK,$TOP,ENTRY),
            ISPEINGDEFINED))
        THEN
        BEGIN
            ERRMS "THIS ID IS NOT A FORMAL PARAMETER.";
            WRITEMESOUT
        END
        ELSE
        BEGIN
            IF CURRENTDECLTYPE = STRUCT
            THEN
            BEGIN
                $ENTER(IDENTRY,$ALLOF(IDSTACK,$TOP));
                $POP(IDSTACK);
                $ENTER(IDSTACK,$OPD(RIGHTAR,
                    $OPD(MSTACK,$TOP,ENTRY),
                    IDSTACKPTR),
                    NUMBRSUBSTRUCT:$OPD(IDENTRY,NUMBRSUBSTRUCT),
                    STRUCTUREPTR:$OPD(IDENTRY,STRUCTUREPTR),
                    STRUCTPTRBEFORE:$OPD(IDENTRY,STRUCTPTRBEFORE));
            END;
            $ENTER(IDSTACK,$OPD(RIGHTAR,
                $OPD(MSTACK,$TOP,ENTRY),
                IDSTACKPTR),
                TYPE:
                MACRO:
                PROCEDURE:
                HASPEENSPEC:
                1,
                1);
            IF CURRENTDECLTYPE ≠ STRUCT
            THEN
            BEGIN
                $ENTER(IDSTACK,$OPD(MSTACK,$TOP,ENTRY),
                    STRUCTPTRBEFORE: PTSTRUCTURETAB)
            END
        END
    END
    ELSE
    BEGIN

```

```

00804700
00804800
00804900
00805000
00805100
00805200
00805300
00805400
00805500
00805600
00805700
00805800
00805900
00806000
00806100
00806200
00806300
00806400
00806500
00806600
00806700
00806800
00806900
00807000
00807100
00807200
00807300
00807400
00807500
00807600
00807700
00807800
00807900
00808000
00808100
00808200
00808300
00808400
00808500
00808600
00808700
00808800
00808900
00809000
00809100
00809200
00809300
00809400
00809500
00809600
00809700
00809800
00809900
00810000
00810100
00810200
00810300
00810400
00810500
00810600
00810700

```

```

IF CURRENTDECLTYPE ≠ STRUCT                                00810800
THEN                                                         00810900
BEGIN                                                         00811000
    $PUSH(IDSTACK, $ALL 0: 0,                                00811100
           $ALL 1: 0,                                         00811200
           $ALL 2: 0,                                         00811300
           $ALL 3: 0,                                         00811400
           TYPE: CURRENTDECLTYPE,                             00811500
           STRUCTPTRBEFORE: PTSTRUCTUREPTR,                  00811600
           MACRO: CURRENTDECLMACRO);                          00811700
END;                                                         00811800
$ENTER(IDSTACK, $TOP, BIGTABPTR: $OPD(MSTACK, $TOP, ENTRY), 00811900
        PROCEDURE: 1,                                         00812000
        OLDSEMFIELD: $OPD(BIGTAB,                             00812100
                           $OPD(MSTACK, $TOP, ENTRY),         00812200
                           SEM));                             00812300
$ENTER(BIGTAB, $OPD(MSTACK, $TOP, ENTRY),                    00812400
        ISDEFINED: 0,                                         00812500
        ISBEINGDEFINED: 1,                                    00812600
        IDSTACKPTR: PTIDSTACK-1);                             00812700
$INCR(IDCOUNTER);                                           00812800
IF $OPD(IDSTACK, $TOP, ISBEINGDEFINED) = 1                  00812900
AND NOT BOOLEAN($OPD(IDSTACK, $TOP, FORWARDDEFINED))        00813000
AND NOT SCANNINGFORMALPARAMETERDECLARATIONS                 00813100
THEN                                                         00813200
BEGIN                                                         00813300
    ERRMS "THIS PROCEDURE IDENTIFIER HAS BEEN USED PREVIOUSLY IN", 00813400
    " THIS BLOCK HEAD. THE PREVIOUS IDENTIFIER IS LOST.";    00813500
    WRITEMESOUT                                              00813600
END;                                                         00813700
END;                                                         00813800
$PUSH(PROCEDUREIDSTACK, CURRENTPROCEDUREID);                00813900
CURRENTPROCEDUREID + $OPD(BIGTAB, $OPD(MSTACK, $TOP, ENTRY), 00814000
                           IDSTACKPTR);                      00814100
$EXEC(SCANSUBSTRUCTURE);                                     00814200
END;                                                         00814300
$ACTION(PROCESSDECLARATION):                                00814400
    COMMENT THIS ACTION SETS CURRENTDECLTYPE = PROS TO INDICATE 00814500
    THAT A PROCESS DECLARATION IS BEING SCANNED;              00814600
    CURRENTDECLTYPE + PROS;                                    00814700
$ACTION(PUSHDUMMYID):                                       00814800
    COMMENT THIS ACTION PUSHES AN ID ENTRY INTO IDSTACK AND DOES 00814900
    NOTHING ELSE. IT IS USED TO PLANT AN ID FOR STRUCTURE PROCEDURES; 00815000
    DROP                                                       00815100
    $PUSH(IDSTACK, $ALL 0: 0,                                00815200
           $ALL 1: 0,                                         00815300
           $ALL 2: 0,                                         00815400
           $ALL 3: 0,                                         00815500
           OWN: CURRENTDECLOWN,                               00815600
           TYPE: CURRENTDECLTYPE,                             00815700
           NODE: CURRENTDECLNODE,                             00815800
           STRUCTPTRBEFORE: PTSTRUCTUREPTR,                  00815900
           MACRO: CURRENTDECLMACRO);                          00816000
END;                                                         00816100
$ACTION(RESETIDLISTCOUNTER):                                00816200
    COMMENT AT THE END OF AN INITIALIZATION IN A DECLARATION, WE 00816300
    MUST AT LEAST SET THE ID LIST COUNTER BACK TO ZERO ;      00816400
    IDLISTCOUNTER + 0;                                         00816500
$ACTION(RESETSPT                                           00816600
        );                                                    00816700
    COMMENT THIS ACTION RESETS CURRENTSTRUCTUREPTR AFTER A     00816800
    STRUCTURE HAS BEEN COMPLETELY SCANNED;

```

```

CURRENTSTRUCTUREPTR + 0;                                00816900
$ACTION(SCANNINGFORMALPARAMETERDECLARATIONS);          00817000
    COMMENT AT THE END OF A FORMAL PARAMETER LIST, THIS ACTION IS 00817100
    USED TO TURN ON THE SCANNINGFORMALPARAMETERDECLARATIONS SWITCH. 00817200
    THIS SWITCH SIGNALS THE DECLARATION PROCESSORS THAT THERE IS 00817300
    SPECIAL PROCESSING REQUIRED;                                00817400
    DROP                                                    00817500
    $PUSH(FORMALSWITCHSTACK;SCANNINGFORMALPARAMETERDECLARATIONS); 00817600
    SCANNINGFORMALPARAMETERDECLARATIONS + TRUE;          00817700
    END;                                                    00817800
$ACTION(SCANPTRFUNCTION);                                00817900
$ACTION(SCANRELEASE);                                    00818000
$ACTION(SCANSUBSTRUCTURE);                               00818100
    COMMENT THIS ACTION PREPARES FOR THE SCAN OF A DECLARATION LIST 00818200
    IN A STRUCTURE DECLARATION. ALL DECLARATION PARAMETERS ARE PUSHED 00818300
    INTO STRUCTUREPARAMSTACK AND THEN SET TO ZERO. ALL BIGTAR 00818400
    IDENTIFIERS ARE SET TO DEFINED SO AS TO PREVENT THEIR BEING 00818500
    EXCLUDED FROM USE WITHIN THE STRUCTURE. IN EFFECT, WE PERFORM A 00818600
    BLOCK END OPERATION AND REINITIALIZE AS IF WE WERE IN A NEW BLOCK; 00818700
    DROP                                                    00818800
    $PUSH(STRUCTUREPARAMSTACK;CURRENTDECLDOWN);          00818900
    $PUSH(STRUCTUREPARAMSTACK;IDLISTCOUNTER);            00819000
    $PUSH(STRUCTUREPARAMSTACK;CURRENTDECLNODE);          00819100
    $PUSH(STRUCTUREPARAMSTACK;CURRENTDECLMACRO);         00819200
    $PUSH(STRUCTUREPARAMSTACK;CURRENTDECLTYPE);          00819300
    $PUSH(FORMALSWITCHSTACK;SCANNINGFORMALPARAMETERDECLARATIONS); 00819400
    SCANNINGFORMALPARAMETERDECLARATIONS + FALSE;         00819500
    IDLISTCOUNTER + 0;                                     00819600
    $EXEC(ENDOFDECLIST);                                   00819700
    $EXEC(ENDOFDATADECLARATION);                          00819800
    END;                                                    00819900
$ACTION(SETCONVERTDEFAULT);                              00820000
$ACTION( SETCURRENTSTRUCTUREPTR                          ) ; 00820100
    COMMENT THIS ACTION SETS UP CURRENTSTRUCTUREPTR AND 00820200
    CURRENTNUMBEROF SUBSTRUCTURES FOR A STRUCTURE FIRST PART; 00820300
    DROP                                                    00820400
    IF CURRENTSTRUCTUREPTR = 0                               00820500
    THEN                                                    00820600
    BEGIN                                                  00820700
        I + $OPD(BIGTAR,$OPD(MSTACK,$TOP,ENTRY),IDSTACKPTR); 00820800
        $OPD(MSTACK,$TOP,STRUCTUREPTR) +                  00820900
        CURRENTSTRUCTUREPTR + $OPD(IDSTACK,I,STRUCTUREPTR); 00821000
        $OPD(MSTACK,$TOP,NUMBRSUBSTRUCT) +                00821100
        CURRENTNUMBEROF SUBSTRUCTURES + $OPD(IDSTACK,I,NUMBRSUBSTRUCT); 00821200
    END                                                    00821300
    ELSE                                                  00821400
    BEGIN                                                  00821500
        $OPD(MSTACK,$TOP,STRUCTUREPTR) +                  00821600
        CURRENTSTRUCTUREPTR +                              00821700
        $OPD(STRUCTURETAR,CURRENTIDPTR,STRUCTUREPTR);     00821800
        $OPD(MSTACK,$TOP,NUMBRSUBSTRUCT) +                00821900
        CURRENTNUMBEROF SUBSTRUCTURES +                    00822000
        $OPD(STRUCTURETAR,CURRENTIDPTR,NUMBRSUBSTRUCT); 00822100
    END;                                                  00822200
    NONAMECALL + TRUE;                                     00822300
    END;                                                  00822400
$ACTION(SETDEFAULTSTRINGLENGTH);                         00822500
$ACTION(SETDEFAULTSTRINGLENGTHANDSIZE);                  00822600
$ACTION(SETDEFAULTSTRINGSIZE);                           00822700
$ACTION(SETFIXEDSTRINGLENGTH);                           00822800
$ACTION(SETSTAROK);                                      00822900

```

```

    ASTERISKALLOWED + TRUE;
$ACTION(SETSTRINGSIZE);
$ACTION(SETUPACCESSPARAMETERSCAN);
    COMMENT THIS ACTION PREPARES FOR AN ACCESS PARAMETER SCAN IN
    AN ACCESS DECLARATION;
DROP
    $PUSH(ACTUALPARMSTACK,CURRENTFORMALPARAMETERLISTPTR);
    CURRENTFORMALPARAMETERLISTPTR +
        $OPD(IDSTACK,CURRENTACCESSID,FORMALPARMPTR);
    $PUSH(ACTUALPARMSTACK,CURRENTNUMBEROFFORMALPARAMETERS);
    CURRENTNUMBEROFFORMALPARAMETERS +
        $OPD(IDSTACK,CURRENTACCESSID,NUMPROFPARMS);
    $PUSH(ACTUALPARMSTACK,CURRENTACTUALPARAMETERCOUNT);
    CURRENTACTUALPARAMETERCOUNT + 0;
    IF CURRENTNUMBEROFFORMALPARAMETERS = 0
    THEN
    BEGIN
        ERRMS POINTER($OPD(BIGTAB,(I+$OPD(IDSTACK,
            CURRENTACCESSID,BIGTABPTR)+1),$ALL))
            FOR $OPD(BIGTAB,I,INLENGTH),
            " ACCESS DECLARATIONS DO NOT HAVE PARAMETERS.";
        WRITEMESOUT
    END
END;
$ACTION(SETUPACTUALPARAMETERSCAN);
    COMMENT THIS ACTION PREPARES FOR AN ACTUAL PARAMETER SCAN BY
    PUSHING AND RESETING THE APPROPRIATE FORMAL AND ACTUAL PARAMETER
    POINTERS AND COUNTERS ;
DROP
    $PUSH(ACTUALPARMSTACK,CURRENTFORMALPARAMETERLISTPTR);
    CURRENTFORMALPARAMETERLISTPTR +
        IF BOOLEAN($OPD(MSTACK,$TOP-1,PROCEDUREPTRISINTOIDSTACK))
        THEN $OPD(IDSTACK,
            $OPD(MSTACK,$TOP-1,PROCEDUREPTR),
            FORMALPARMPTR)
        ELSE $OPD(STRUCTURETAB,
            $OPD(MSTACK,$TOP-1,PROCEDUREPTR),
            FORMALPARMPTR);
    $PUSH(ACTUALPARMSTACK,CURRENTNUMBEROFFORMALPARAMETERS);
    CURRENTNUMBEROFFORMALPARAMETERS +
        IF BOOLEAN($OPD(MSTACK,$TOP-1,PROCEDUREPTRISINTOIDSTACK))
        THEN $OPD(IDSTACK,
            $OPD(MSTACK,$TOP-1,PROCEDUREPTR),
            NUMPROFPARMS)
        ELSE $OPD(STRUCTURETAB,
            $OPD(MSTACK,$TOP-1,PROCEDUREPTR),
            NUMPROFPARMS);
    $PUSH(ACTUALPARMSTACK,CURRENTACTUALPARAMETERCOUNT);
    CURRENTACTUALPARAMETERCOUNT + 0;
    IF CURRENTNUMBEROFFORMALPARAMETERS = 0
    THEN
    BEGIN
        ERRMS "THIS PROCEDURE MAY NOT HAVE PARAMETERS." ;
        WRITEMESOUT
    END;
END;
COMMENT
    THESE ACTIONS ALL TEST THE TYPE OF THE NEXT FORMAL PARAMETER;
$ACTION(SETVARIABLESTRINGLENGTH);
$ACTION(SKIPTHROUGHSEMICOLON);
DROP

```

00823000
00823100
00823200
00823300
00823400
00823500
00823600
00823700
00823800
00823900
00824000
00824100
00824200
00824300
00824400
00824500
00824600
00824700
00824800
00824900
00825000
00825100
00825200
00825300
00825400
00825500
00825600
00825700
00825800
00825900
00826000
00826100
00826200
00826300
00826400
00826500
00826600
00826700
00826800
00826900
00827000
00827100
00827200
00827300
00827400
00827500
00827600
00827700
00827800
00827900
00828000
00828100
00828200
00828300
00828400
00828500
00828600
00828700
00828800
00828900
00829000

```

DO SKAN(1) UNTIL $OPD(MSTACK,PTMSTACK,PSYM) = "}")
END)
$ACTION(STARNOTALLOWED):
    ASTERISKALLOWED + FALSE)
$ACTION(STARTSUBSCRIPT
    COMMENT THIS ACTION PREPARES TO SCAN A SUBSCRIPT LIST BY
    PUSHING AND CLEARING CURRENTNUMBROFSUBS;
    DROP
    $PUSH(SUBSCRIPTSTACK;CURRENTNUMBROFSUBSTRUCTURES);
    $PUSH(SUBSCRIPTSTACK;CURRENTSTRUCTUREPTR);
    $PUSH(SUBSCRIPTSTACK;CURRENTNUMBROFSUBS);
    CURRENTNUMBROFSUBS + CURRENTSTRUCTUREPTR + 0;
    END)
$ACTION(STRINGDECLARATION):
    COMMENT THIS ACTION SETS CURRENTDECLTYPE = STR TO INDICATE
    THAT A STRING DECLARATION IS BEING SCANNED;
    CURRENTDECLTYPE + STR;
$ACTION(STRINGPROCEDURE):
    DROP
    SEMANTICTEST + (IF CURRENTSTRUCTUREPTR = 0
        THEN $OPD(IDSTACK,CURRENTPROCEDUREID,TYPE)
        ELSE $OPD(STRUCTURETAB,CURRENTIDPTR,TYPE))
        = STR )
    END)
$ACTION(STRINGSTORE):
    DROP
    OPERATOR("STORE=STRING");
    END)
$ACTION(STRUCTUREDECLARATION):
    COMMENT THIS ACTION SETS CURRENTDECLTYPE AND STRUCTUREISOWN
    FOR LATER USE IN A STRUCTURE DECLARATION;
    STRUCTUREISOWN + CURRENTDECLOWN = 1;
    CURRENTDECLTYPE + STRUCT;
$ACTION(STRUCTUREMUSTBENAMEDBASE):
    COMMENT THIS ACTION PRINTS AN ERROR MESSAGE IF THE STRUCTURE
    JUST DECLARED WAS NOT NAMED "BASE". THIS ONLY OCCURS IN ACCESS
    DEFINITIONS;
    DROP
    IF $OPD(BIGTAB,(I+$OPD(IDSTACK,$TOP,BIGTABPTR)),IDLENGTH) # 4
        OR POINTER($OPD(BIGTAB,I+1,$ALL)) # "BASE"
    THEN
    BEGIN
        ERRMS "THE FIRST STRUCTURE IN AN ACCESS DEFINITION MUST BE ",
            "NAMED ", "", "BASE", "", "", ".";
        WRITEMESOUT;
    END)
    END)
$ACTION(STRUCTUREMUSTBENAMEDENTRY):
    COMMENT THIS ACTION PRINTS AN ERROR MESSAGE IF THE STRUCTURE
    JUST DECLARED WAS NOT NAMED "ENTRY". THIS ONLY OCCURS IN ACCESS
    DEFINITIONS;
    DROP
    IF $OPD(BIGTAB,(I+$OPD(IDSTACK,$TOP,BIGTABPTR)),IDLENGTH) # 5
        OR POINTER($OPD(BIGTAB,I+1,$ALL)) # "ENTRY"
    THEN
    BEGIN
        ERRMS "THE SECOND STRUCTURE IN AN ACCESS DEFINITION MUST",
            ", BE NAMED ", "", "ENTRY", "", "", ".";
        WRITEMESOUT
    END
    END)
END)

```

[illegible]

OSL2 /PGM1 LISTED BY THESIS ON JANUARY 3, 1971 AT 12:13 P.M.

BEGIN	00000100
SUBJECT	00000200
INTEGER I,J,(K,L,M)+0,X,Y;	00000300
POINTER P+ NULL;	00000400
BOOLEAN TEST1,TEST2;	00000500
STRUCTURE JOR(	00000600
INTEGER NUMBER,PRIORITY;	00000700
STRUCTURE RECORDS(	00000800
INTEGER TIMESTARTED,TIMEFINISHED,	00000900
NUMEROFLINES);	00001000
POINTER NEXT,LAST);	00001100
I + J + (K + **1)*I + J/K;	00001200
TEST1 + TEST2 + FALSE;	00001300
JOB.NUMBER + IF TEST1 THEN 0 ELSE J FI;	00001400
BEGIN	00001500
INTEGER Z,Q + K+1 - IF TEST1 THEN 0 ELSE L+M FI;	00001600
BOOLEAN TEST2,TEST3;	00001700
TEST1 + TEST2 + TEST3 + TRUE;	00001800
X + JOB.NUMBER + JOB.RECORDS.TIMEFINISHED	00001900
- JOB.RECORDS.TIMESTARTED;	00002000
Z + IF TEST2 AND TEST1	00002100
THEN IF Q >= 3 THEN L ELSE M FI	00002200
ELSE CASE J OF K,K,K,L,L,L,M,M,M,	00002300
IF I + (**1) MOD 50 = 2	00002400
THEN I ELSE 0 FI	00002500
ESAC	00002600
FI;	00002700
TEST1 + TEST2 + FALSE	00002800
END;	00002900
TEST1 + TEST2 + TRUE	00003000
END	00003100

ILLIAC IV TRANSLATOR WRITING SYSTEM

OSL2 COMPILER - VERSION 1

JANUARY 3, 1971 AT 12:14 P.M.

PROGRAM PAA /PAA

REGI:	00000100	1
SUBJECT	00000200	2
INTEGER I,J,(K,I,M)+0,X,Y)	00000300	3
<pre> (00,0000) = 1 (00,0001) = J (00,0002) = K (00,0003) = I (00,0004) = M 00100000 NAME-CALL (00,0002) 00100001 NAME-CALL (00,0003) 00100002 NAME-CALL (00,0004) 00100003 INTEGER-LITERAL-CALL 0 00100004 STOP-INTEGER-PARTIAL-DESTRUCTIVE 00100005 STOP-INTEGER-PARTIAL-DESTRUCTIVE 00100006 STOP-INTEGER (00,0005) = 0 (00,0006) = Y POINTER P+ NULL; (00,0007) = P 00100007 NAME-CALL (00,0007) 00100008 NULL 00100009 STOP-POINTER BOOLEAN TEST1,TEST2; (00,0008) = TEST1 (00,0009) = TEST2 STRUCTURE JOP(INTEGER NUMBER,PRIORITY) (00,0010) = JOP (00,0011) = NUMBER (00,0012) = PRIORITY STRUCTURE RECORDS(INTEGER TIMESTARTED,TIMEFINISHED, (00,0013) = RECORDS (00,0014) = TIMESTARTED (00,0015) = TIMEFINISHED NUMEROFLINES)) (00,0016) = NUMEROFLINES POINTER NEXT,LAST)) (00,0014) = NEXT </pre>		
	00000400	4
	00000500	5
	00000600	6
	00000700	7
	00000800	8
	00000900	9
	00001000	10
	00001100	11

```

(00,0015) = LAST
I ← J + (K + **1)*I + J/K;                                00001200      12

001:0010      NAME-CALL (00,0000) I
001:0011      NAME-CALL (00,0001) J
001:0012      NAME-CALL (00,0002) K
001:0013      DUPLICATE-TOP-OF-STACK-AND-LOAD-INTEGER
001:0014      INTEGER-LITERAL-CALL 1
001:0015      ADD
001:0016      STORE-INTEGER
001:0016***   STORE-INTEGER-PARTIAL-DESTRUCTIVE
001:0017      NAME-CALL (00,0000) I
001:0018      LOAD-INTEGER
001:0019      MULTIPLY
001:0020      NAME-CALL (00,0001) J
001:0021      LOAD-INTEGER
001:0022      NAME-CALL (00,0002) K
001:0023      LOAD-INTEGER
001:0024      DIVIDE
001:0025      ADD
001:0026      STORE-INTEGER
001:0026***   STORE-INTEGER-PARTIAL-DESTRUCTIVE
001:0027      STORE-INTEGER
TEST1 ← TEST2 ← FALSE;                                00001300      13

001:0028      NAME-CALL (00,0008) TEST1
001:0029      NAME-CALL (00,0009) TEST2
001:0030      FALSE
001:0031      STORE-BOOLEAN
001:0031***   STORE-BOOLEAN-PARTIAL-DESTRUCTIVE
001:0032      STORE-BOOLEAN
JOB.NUMBER ← IF TEST1 THEN 0 ELSE J FI;                00001400      14

001:0033      NAME-CALL (00,0010) JOB
001:0034      INTEGER-LITERAL-CALL 000
001:0035      MAKE-INDEX NUMBER
001:0036      NAME-CALL (00,0008) TEST1
001:0037      LOAD-BOOLEAN
001:0038      *** (BRANCH-IF-FALSE INSTRUCTION WILL BE HERE) ***
001:0039      INTEGER-LITERAL-CALL 0
001:0038***   BRANCH-IF-FALSE (001:0041)
001:0040      *** (BRANCH INSTRUCTION WILL BE HERE) ***
001:0041      NAME-CALL (00,0001) J
001:0042      LOAD-INTEGER
001:0040***   BRANCH (001:0043)
001:0043      STORE-INTEGER
BEGIN                                                    00001500      15

INTEGER Z ← 0 ← K+1 - IF TEST1 THEN 0 ELSE L+M FI;    00001600      16

(01,0000) = 7
(01,0001) = 0
002:0000      NAME-CALL (01,0001)
002:0001      NAME-CALL (00,0002) K
002:0002      LOAD-INTEGER
002:0003      INTEGER-LITERAL-CALL 1
002:0004      ADD
002:0005      NAME-CALL (00,0008) TEST1
002:0006      LOAD-BOOLEAN
002:0007      *** (BRANCH-IF-FALSE INSTRUCTION WILL BE HERE) ***
002:0008      INTEGER-LITERAL-CALL 0

```

```

002:0007*** BRANCH-IF-FALSE (002:0010)
002:0009 *** (BRANCH INSTRUCTION WILL BE HERE) ***
002:0010 NAME-CALL (00,0003) L
002:0011 LOAD-INTEGER
002:0012 NAME-CALL (00,0004) M
002:0013 LOAD-INTEGER
002:0014 ADD
002:0009*** BRANCH (002:0015)
002:0015 SUBTRACT
002:0016 STORE-INTEGER
BOOLEAN TEST2,TEST3 00001700 17

(01,0002) = TEST2
(01,0003) = TEST3
TEST1 ← TEST2 + TEST3 + TRUE 00001800 18

002:0017 NAME-CALL (00,0008) TEST1
002:0018 NAME-CALL (01,0002) TEST2
002:0019 NAME-CALL (01,0003) TEST3
002:0020 TRUE
002:0021 STORE-BOOLEAN
002:0021*** STORE-BOOLEAN-PARTIAL-DESTRUCTIVE
002:0022 STORE-BOOLEAN
002:0022*** STORE-BOOLEAN-PARTIAL-DESTRUCTIVE
002:0023 STORE-BOOLEAN
X ← JOB.NUMBER + JOB.RECORDS.TIMEFINISHED 00001900 19

002:0024 NAME-CALL (00,0005) X
002:0025 NAME-CALL (00,0010) JOB
002:0026 INTEGER-LITERAL-CALL 000
002:0027 MAKE-INDEX NUMBER
002:0026 LOAD-INTEGER
002:0029 NAME-CALL (00,0010) JOB
002:0030 INTEGER-LITERAL-CALL 002
002:0031 MAKE-INDEX RECORDS
002:0032 INTEGER-LITERAL-CALL 001
002:0033 MAKE-INDEX TIMEFINISHED
- JOB.RECORDS.TIMESTARTED; 00002000 20

002:0034 LOAD-INTEGER
002:0035 ADD
002:0036 NAME-CALL (00,0010) JOB
002:0037 INTEGER-LITERAL-CALL 002
002:0038 MAKE-INDEX RECORDS
002:0039 INTEGER-LITERAL-CALL 000
002:0040 MAKE-INDEX TIMESTARTED
002:0041 LOAD-INTEGER
002:0042 SUBTRACT
002:0043 STORE-INTEGER
Z ← IF TEST2 AND TEST1 00002100 21

002:0044 NAME-CALL (01,0000) Z
002:0045 NAME-CALL (01,0002) TEST2
002:0046 LOAD-BOOLEAN
THEN IF 0 ≥ 3 THEN L ELSE M FI 00002200 22

002:0047 NAME-CALL (00,0008) TEST1
002:0048 LOAD-BOOLEAN
002:0049 AND
002:0050 *** (BRANCH-IF-FALSE INSTRUCTION WILL BE HERE) ***
002:0051 NAME-CALL (01,0001) 0

```

```

002:0052  LOAD-INTEGER
002:0053  INTEGER-LITERAL-CALL 3
002:0054  INTEGER-GREATER-OR-EQUAL
002:0054  *** (BRANCH-IF-FALSE INSTRUCTION WILL BE HERE) ***
002:0055  NAME-CALL (00,0003) L
002:0056  LOAD-INTEGER
002:0054*** BRANCH-IF-FALSE (002:0058)
002:0057  *** (BRANCH INSTRUCTION WILL BE HERE) ***
002:0058  NAME-CALL (00,0004) M
           ELSE CASE J OF K,K,K,L,L,L,M,M,M,

```

00002300

23

```

002:0059  LOAD-INTEGER
002:0057*** BRANCH (002:0060)
002:0050*** BRANCH-IF-FALSE (002:0061)
002:0060  *** (BRANCH INSTRUCTION WILL BE HERE) ***
002:0061  NAME-CALL (00,0001) J
002:0062  LOAD-INTEGER
002:0063  *** (INDEXED-BRANCH INSTRUCTION WILL BE HERE) ***
002:0064  NAME-CALL (00,0002) K
002:0065  LOAD-INTEGER
002:0066  *** (BRANCH INSTRUCTION WILL BE HERE) ***
002:0067  NAME-CALL (00,0002) K
002:0068  LOAD-INTEGER
002:0069  *** (BRANCH INSTRUCTION WILL BE HERE) ***
002:0070  NAME-CALL (00,0002) K
002:0071  LOAD-INTEGER
002:0072  *** (BRANCH INSTRUCTION WILL BE HERE) ***
002:0073  NAME-CALL (00,0003) L
002:0074  LOAD-INTEGER
002:0075  *** (BRANCH INSTRUCTION WILL BE HERE) ***
002:0076  NAME-CALL (00,0003) L
002:0077  LOAD-INTEGER
002:0078  *** (BRANCH INSTRUCTION WILL BE HERE) ***
002:0079  NAME-CALL (00,0003) L
002:0080  LOAD-INTEGER
002:0081  *** (BRANCH INSTRUCTION WILL BE HERE) ***
002:0082  NAME-CALL (00,0004) M
002:0083  LOAD-INTEGER
002:0084  *** (BRANCH INSTRUCTION WILL BE HERE) ***
002:0085  NAME-CALL (00,0004) M
002:0086  LOAD-INTEGER
002:0087  *** (BRANCH INSTRUCTION WILL BE HERE) ***
002:0088  NAME-CALL (00,0004) M
           IF I + (**+1) MOD 50 = 2

```

00002400

24

```

002:0089  LOAD-INTEGER
002:0090  *** (BRANCH INSTRUCTION WILL BE HERE) ***
002:0091  NAME-CALL (00,0000) T
002:0092  DUPLICATE-TOP-OF-STACK-AND-LOAD-INTEGER
002:0093  INTEGER-LITERAL-CALL 1
002:0094  ADD
002:0095  INTEGER-LITERAL-CALL 50
002:0096  MOD
002:0097  STOP-INTEGER
002:0097*** STOP-INTEGER-PARTIAL-DESTRUCTIVE
002:0098  INTEGER-LITERAL-CALL 2
           THEN I ELSE 0 FI

```

00002500

25

```

002:0099  INTEGER-EQUAL
002:0099  *** (BRANCH-IF-FALSE INSTRUCTION WILL BE HERE) ***
002:0100  NAME-CALL (00,0000) I

```

002:0101	LOAD=INTEGER		
002:0099***	BRANCH=IF=FALSE	(002:0103)	
002:0102	*** (BRANCH INSTRUCTION WILL BE HERE) ***		
002:0103	INTEGER=LITERAL-CALL	0	
002:0102***	BRANCH	(002:0104)	
	ESAC		00002600 26
002:0104	*** (BRANCH INSTRUCTION WILL BE HERE) ***		
002:0105***	BRANCH	(002:0064)	
002:0066***	BRANCH	(002:0115)	
002:0106***	BRANCH	(002:0067)	
002:0069***	BRANCH	(002:0115)	
002:0107***	BRANCH	(002:0070)	
002:0072***	BRANCH	(002:0115)	
002:0108***	BRANCH	(002:0073)	
002:0075***	BRANCH	(002:0115)	
002:0109***	BRANCH	(002:0076)	
002:0078***	BRANCH	(002:0115)	
002:0110***	BRANCH	(002:0079)	
002:0081***	BRANCH	(002:0115)	
002:0111***	BRANCH	(002:0082)	
002:0084***	BRANCH	(002:0115)	
002:0112***	BRANCH	(002:0085)	
002:0087***	BRANCH	(002:0115)	
002:0113***	BRANCH	(002:0088)	
002:0090***	BRANCH	(002:0115)	
002:0114***	BRANCH	(002:0091)	
002:0104***	BRANCH	(002:0115)	
002:0063***	INDEXED-BRANCH	(002:0105)	
	FIJ		00002700 27
002:0060***	BRANCH	(002:0115)	
002:0115	STORE=INTEGER		
	TEST1 + TEST2 + FALSE		00002800 28
002:0116	NAME-CALL	(00,0008) TEST1	
002:0117	NAME-CALL	(01,0002) TEST2	
002:0118	FALSE		
	ENDJ		00002900 29
002:0119	STORE=BOOLEAN		
002:0119***	STORE=BOOLEAN-PARTIAL-DESTRUCTIVE		
002:0120	STORE=BOOLEAN		
	TEST1 + TEST2 + TRUE		00003000 30
001:0044	NAME-CALL	(00,0008) TEST1	
001:0045	NAME-CALL	(00,0009) TEST2	
001:0046	TRUE		
	END		00003100 31
001:0047	STORE=BOOLEAN		
001:0047***	STORE=BOOLEAN-PARTIAL-DESTRUCTIVE		
001:0048	STORE=BOOLEAN		

CONGRATULATIONS, NO SOURCE PROGRAM ERRORS WERE DETECTED IN PASS 1.

THE TOTAL CPU TIME WAS	0 MINUTES 30.0 SECONDS.
THE TOTAL I/O TIME WAS	0 MINUTES 37.5 SECONDS.
THE TOTAL ELAPSED TIME WAS	1 MINUTES 9.7 SECONDS.

LIST OF REFERENCES

- [1] Attached Support Processor System (ASP) System Programmers Manual, H20-0323, IBM Technical Publications Department, White Plains, N.Y., 1968.
- [2] Attached Support Processor System (ASP) System Description, H20-0466, IBM Technical Publications Department, White Plains, N.Y., 1968.
- [3] Attached Support Processor System (ASP) Applications Programmer Manual, H20-0322, IBM Technical Publications Department, White Plains, N.Y., 1967.
- [4] Attached Support Processor System (ASP) Console Operators Manual, H20-0321, IBM Technical Publications Department, White Plains, N.Y., 1968.
- [5] HASP-II, PID#360D-05.1.014, IBM Program Information Department, Hawthorn, N.Y., 1969.
- [6] IBM 7090/7094 IBSYS Operating System Version 13: System Monitor (IBSYS), C28-6248, IBM Technical Publications Department, White Plains, N.Y.
- [7] PORTHOS User's Manual, Department of Computer Science, University of Illinois, Urbana, Illinois, 1967.
- [8] A Narrative Description of the Burroughs B5500 Diskfile Master Control Program, Burroughs Corporation, Detroit, 1966.
- [9] ESPOL Reference Manual, Burroughs Corporation, Detroit, 1966.
- [10] E. W. Dijkstra, The Structure of the "THE" - Multiprogramming System, Comm. of the Assoc. for Computing Mach., v.11, no.5, May, 1968.
- [11] P. A. Alsberg and R. A. Wells, OSL-An Operating System Language, CS 497 special project, University of Illinois, copies available from authors, May 24, 1968.
- [12] P. A. Alsberg and C. R. Mills, The Structure of the ILLIAC IV Operating System, Proceedings of the Second Symposium on Operating System Principles, Princeton University, October 20-22, 1969.
- [13] P. A. Alsberg, et al, A Description of the ILLIAC IV Operating System, ILLIAC IV Document No. 212, University of Illinois, November, 1968.
- [14] E. W. Dijkstra, Co-operating Sequential Processes, Math. Dept., Tech. University, Eindhoven, 1965.

- [15] A. N. Habermann, On the Harmonious Co-operation of Abstract Machines, Ph.D. Thesis, Tech. Univ., Eindhoven, 1967.
- [16] J. V. Garwick, GPL, A Truly General Purpose Language, Comm. of the Assoc. for Computing Mach., v.11, no.9, September, 1968.
- [17] J. V. Garwick, GPL, A General Purpose Language, SIGPLAN Notices, v.4, no.8, August, 1969.
- [18] A. V. Wijngaarden, et al, Report on the Algorithmic Language Algol 68, Math. Cent., Amsterdam, MR101, February, 1969.
- [19] B. J. Mailloux and J. E. L. Peck, Algol 68 as an Extensible Language, SIGPLAN Notices, v.4, no.8, August, 1969.
- [20] R. M. Balzer, EXDAMS-EXTendable Debugging and Monitoring System, AFIPS Conference Proceedings - Spring Joint Computer Conference, v.34, AFIPS Press, Montvale, N.J., May, 1969, pp. 567-580.
- [21] B6500 Reference Manual, 1043676, Burroughs Corporation, Detroit, 1969.
- [22] W. A. Wulf, et al, BLISS Reference Manual, Department of Computer Science, Carnegie Mellon University, Pittsburgh, 1970.
- [23] B5500 EXTENDED ALGOL Reference Manual, 1028024, Burroughs Corporation, Detroit, 1969.
- [24] B6700 EXTENDED ALGOL Language Information Manual, 5000128, Burroughs Corporation, Detroit, 1971.
- [25] A. J. Beals, J. E. LaFrance, and R. S. Northcote, The Automatic Generation of Floyd Production Syntactic Analyzers, Department of Computer Science Report No. 350, University of Illinois, Urbana, Illinois, 1969.
- [26] R. L. Mercer, TWINKLE-A Syntax Language for a Translator Writing System, Master's Thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1970.
- [27] N. C. Machado, ISL-A Semantics Language for a Translator Writing System, Department of Computer Science Report No. 367, University of Illinois, Urbana, Illinois, 1967.

VITA

Peter Allyn Alsberg received his B.S. in Engineering Physics from the University of Illinois in 1966. His M.S. in Mathematics was completed at the University of Illinois in 1967.

Mr. Alsberg was elected to full membership in Sigma Xi, and the undergraduate honorary Phi Eta Sigma.

While at the University of Illinois, Mr. Alsberg designed and supervised the initial implementation of the supervisor and monitor modules of the ILLIAC IV Operating System. In 1969, together with Mr. C. R. Mills, he published "The Structure of the ILLIAC IV Operating System" in the Proceedings of the Second Symposium on Operating System Principles.

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Center for Advanced Computation University of Illinois at Urbana-Champaign Urbana, Illinois 61801		2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	
2b. GROUP			
3. REPORT TITLE OSL/2 AN OPERATING SYSTEM LANGUAGE			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) Research Report			
5. AUTHOR(S) (First name, middle initial, last name) Peter Allyn Alsberg			
6. REPORT DATE June 10, 1971		7a. TOTAL NO. OF PAGES 256	7b. NO. OF REFS 27
8. CONTRACT OR GRANT NO. USAF 30(602)-4144		9a. ORIGINATOR'S REPORT NUMBER(S) CAC Document No. 6	
9b. PROJECT NO. ARPA Order 788		9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report) DCS Report No. 460	
10. DISTRIBUTION STATEMENT Copies may be requested from the address given in (1) above.			
11. SUPPLEMENTARY NOTES None		12. SPONSORING MILITARY ACTIVITY Rome Air Development Center Griffiss Air Force Base Rome, New York 13440	
13. ABSTRACT OSL/2, an Operating System Language, was designed for the coding of supervisory systems. OSL/2 is an ALGOL-based language that may be readily extended to a general purpose language with the addition of a few data types (e.g., floating point numbers). The language replaces an interrupt concept with the P and V operators on semaphores described by Dijkstra. Concepts of multiple processes and system layers allow OSL/2 to be used as a job control language for an OSL/2 coded supervisory system. The system layering concepts provide for easy system extension and hierachical structuring. A powerful method of specifying access techniques is used to implement queues, tables, etc. and treat them as primitive data items in the language. The language is presented in narrative and reference formats. Hardware implications are discussed and an OSL/2 compiler which runs on the Burroughs 5500 is briefly described. The implications of the language on system measurement, simulation, and modification are also discussed.			

UNCLASSIFIED

Security Classification

14.

KEY WORDS

LINK A

LINK B

LINK C

ROLE

WT

ROLE

WT

ROLE

WT

Procedure and Problem Oriented Languages
(Miscellaneous)

Compilers and Generators

Compiler-Compilers

Programming Languages

Supervisory Systems

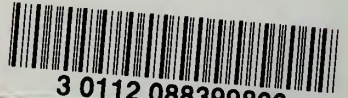
Basic Monitors

UNCLASSIFIED



APR 20 1979

UNIVERSITY OF ILLINOIS-URBANA
510.84 IL6R no. C002 no. 457-462(1971)
Automatic generation of deterministic pe



3 0112 088399800